Learning SQL

修订版

SQL学习指南

（第2版）



[美] *Alan Beaulieu* 著

张伟超 林青松 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

O'REILLY®

SQL 学习指南（第 2 版）（修订版）

[美] Alan Beaulieu　著
张伟超　林青松　译

人 民 邮 电 出 版 社
北 京

图书在版编目（ＣＩＰ）数据

SQL学习指南 ： 第2版 / （美） 比利 （Beaulieu,A.） 著 ； 张伟超， 林青松译. -- 北京 ： 人民邮电出版社， 2010.6
ISBN 978-7-115-22875-8

Ⅰ. ①S… Ⅱ. ①比… ②张… ③林… Ⅲ. ①关系数据库—数据库管理系统，SQL Server Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2010)第073249号

版权声明

SQL 学习指南（第 2 版）

◆ 著 [美] Alan Beaulieu
译 张伟超 林青松
责任编辑 刘映欣
◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷
◆ 开本：787×1092 1/16
印张：19.25
字数：394 千字 2010 年 6 月第 1 版
印数：1 – 000 册 2010 年 6 月北京第 1 次印刷

著作权合同登记号 图字：01-2009-6946 号

ISBN 978-7-115-22875-8

定价： 元

读者服务热线：(010) 67132705 印装质量热线：(010) 67129223
反盗版热线：(010) 67171154

内 容 提 要

本书全面系统地介绍了 SQL 语言各方面的基础知识以及一些高级特性，包括 SQL 数据语言、SQL 方案语言、数据集操作、子查询以及内建函数与条件逻辑等内容。书中每个章节讲述一个相对独立的主题，并提供了相关示例和练习。本书内容以 SQL 92 标准为蓝本，涵盖了市场上常用数据库的最新版本（MySQL 6.0、Oracle 11g 及 Microsoft SQL Server 2008）。

本书适合数据库应用开发者、数据库管理员和高级用户阅读。针对开发基于数据库的应用程序，以及日常的数据库系统管理，本书都展现了大量经过实践检验的方法和技巧。读者可以通过对本书循序渐进地学习快速掌握 SQL 语言，也可以在实际工作中遇到问题时直接翻阅本书中的相关章节以获取解决方案。

O’Reilly Media, Inc.介绍

为了满足你对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O’Reilly Media, Inc.授权人民邮电出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O’Reilly Media, Inc.是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 The Whole Internet User’s Guide & Catalog（被纽约公共图书馆评为 20 世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的 Web 服务器软件），O’Reilly Media, Inc.一直处于 Internet 发展的最前沿。

许多书店的反馈表明，O’Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O’Reilly Media, Inc.具有深厚的计算机专业背景，这使得 O’Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O’Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O’Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O’Reilly Media,Inc.依靠他们及时地推出图书。因为 O’Reilly Media, Inc.紧密地与计算机业界联系着，所以 O’Reilly Media, Inc.知道市场上真正需要什么图书。

作者简介

Alan Beaulieu 从事设计、构建和实现应用数据库已有 15 个年头，他目前经营自己的顾问公司，专门提供金融和电信领域的 Oracle 数据库设计与支持服务。Alan 使用了 Oracle 的诸多特性，如并行查询、分区和并行服务器等，以构建 OLTP 和 OLAP 环境下的大型数据库。Alan 获得了康奈尔大学工程学院的运筹学学士学位，现在和妻子以及两个女儿一起住在马萨诸塞州，可以通过电子邮箱 albeau_mosql@yahoo.com 与他联系。

封面介绍

本书封面上的动物是安第斯有袋树蛙，它的名字暗示了这种在傍晚和夜间活动的青蛙原生地在安第斯山脉的西坡，并且广泛分布在北部的里奥班巴盆地和伊瓦拉之间。

在求偶期，雄蛙发出“哇可-啊可-啊可”的叫声以吸引雌蛙，如果一只怀孕中的雌蛙被吸引过来，雄蛙将会爬到它的背上执行常见的青蛙交配过程。当雌蛙从泄殖腔中排出卵时，雄蛙用脚抓住这些卵并完成受精，然后将它们放到雌蛙的孵化袋中。雌蛙一次可以孵化 130 多枚蛙卵，这些卵需要在它的孵化袋中发育 60～120 天。 在孵化期间，雌蛙的身体将会变得臃肿，并且在它背部皮下将会出现许多肿块。当卵在孵化袋中发育成蝌蚪时，雌性树蛙将它们放入水中。两三个月之后，这些蝌蚪将会变成青蛙，而在 7 个月之后，它们进入了生育期，又开始“哇可-啊可-啊可”了。

雄性和雌性树蛙的前后趾上都进化出许多吸盘，这可以用来帮助它们攀爬树木等垂直的表面。成年雄蛙大小为 2 英寸左右（约 5cm），而雌蛙为 2.5 英寸（6.35cm）。它们的表皮有时为绿色，有时为褐色，有时为这两种颜色的混合色，幼年期的树蛙在生长时会逐渐由褐色变为绿色。

封面图片来自于 *Dover Archive Pictorial*（《多佛尔档案画报》）。

前言

编程语言在不断地出现和消亡，现在使用的语言只有很少一部分的历史能追溯到 20 年前。其中有大量应用在大型机环境的 Cobol 和流行于操作系统、服务器开发以及嵌入式系统的 C 语言。而在数据库领域，SQL 的根源可以追溯到 19 世纪 70 年代。

SQL 是一种从关系型数据库生成、操作和检索数据的语言。关系型数据库流行的原因之一是正确设计的关系型数据库可以处理海量数据。但处理大量数据集时，SQL 就像一个高功率可变焦距的时髦数码相机，让你能够看到大型数据集，或者放大单独的行（或者两者之间的任何地方）。其他的数据库管理系统在沉重的负荷下往往会由于它们的焦距太窄（缩放镜头已经处于最大位置了）而崩溃，这就是要废黜关系型数据库和 SQL 的尝试已经基本上失败了的原因。因此，即使 SQL 是一门古老的语言，它也将继续活跃很长一段时间，并且在存储应用方面有光明的前途。

为什么要学习 SQL

如果打算使用关系型数据库，无论是写应用程序、执行管理任务还是生成报表，那么都需要知道如何与数据库中的数据交互。即使使用工具为自己生成 SQL，比如报表工具，有时也需要绕过自动生成功能而编写自己的 SQL 语句。

学习 SQL 语言有一个额外的好处，即强迫你勇敢面对并学会理解用于储存自己组织相关信息的数据结构。当开始适应数据库的表时，你可能会发现自己也会产生对数据库进行修改或增加等的建议。

为什么使用本书学习 SQL

SQL 语言可分为几类：用于创建数据库对象（表、索引、约束等）的语句统称为 SQL 模式语句，而用于创建、操纵和检索保存在数据库中的数据的语句称为 SQL 数据语句。作为管理员，你将同时使用 SQL 模式和 SQL 数据语句，而程序员或者报表作者可能只需要使用（或者只允许使用）SQL 数据语句。虽然本书介绍了许多 SQL 模式语句，但是主要焦点还是编程功能。

由于只有少数命令，因此 SQL 数据语句看似很简单。依我看来，现在许多 SQL 图书都通过仅仅涉猎这个语言可能的表层知识帮助你培养这种观念。然而，如果打算使用 SQL，那么你就有必要充分理解它的语言能力以及如何组合不同的功能以产生强大的结果。我感觉本书是唯一一本详细介绍 SQL 语言而不会同时被作为门挡的书（正如我知

道的，1250 页的“完全手册”往往被丢在人们的卧室书架上，布满了灰尘）。

虽然本书的示例都可以运行在 MySQL、Oracle 数据库以及 SQL Server 上，但是我必须选择其中之一来作为示例数据库服务器并规范化示例查询返回的结果集。我在这 3 个产品中选择了 MySQL，是因为它可以免费获得、安装简单以及易于管理。对于那些使用其他服务器的人，我建议下载和安装 MySQL 并加载示例数据库，这样就可以运行示例数据库并试验数据了。

本书的结构

本书分为 15 章和 3 个附录。

第 1 章“背景知识”，探讨计算机数据库的历史，其中包括关系模型以及 SQL 语言的出现。

第 2 章“创建和操作数据库”，说明如何创建本书示例使用的 MySQL 数据库和表，以及用数据填充表。

第 3 章“查询入门”，介绍选择语句，然后进一步阐述了大多数常用子句（select、from、where）。

第 4 章“过滤”，说明不同类型的条件，它们可以用于 select、update 或 delete 语句的 where 子句中。

第 5 章“多表查询”，展示如何通过表的连接使用多表进行查询。

第 6 章“使用集合”，介绍所有关于数据集的知识以及它们如何在查询内交互。

第 7 章“数据生成、转换和操作”，说明用于操作或转换数据的几个内置函数。

第 8 章“分组与聚集”，展示如何聚合数据。

第 9 章“子查询”，介绍子查询（个人最喜欢的），并说明如何以及在何处使用它们。

第 10 章“再谈连接”，更加深入地讨论不同类型的表连接。

第 11 章“条件逻辑”，探讨如何在 select、insert、update 和 delete 语句里使用条件逻辑（如 if-then-else）。

第 12 章“事务”，介绍事务及如何使用它们。

第 13 章“索引和约束”，探讨索引和约束。

第 14 章“视图”，说明如何构建接口以屏蔽数据复杂性。

第 15 章“元数据”，说明数据字典的使用。

附录 A“示例数据库的 ER 图”，展示本书所有示例的数据库模式。

附录 B“MySQL 对 SQL 语言的扩展”，说明在 MySQL 的 SQL 实现中一些有趣的非

ANSI 功能。

附录 C“练习答案”，介绍各章习题的答案。

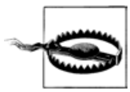
阅读须知

本书使用了下面的印刷约定：



提示

指出提示、建议或者一般注意性的问题。例如，我使用注意向你表明 Oracle 9i 的新功能。



警告

指示一个告诫或提醒。例如，我告诉你如果不小心使用，那么有些 SQL 语句可能会产生意想不到的后果。

联系我们

请将对本书的评论和问题按以下地址与出版社联系。

美国：

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

100035 北京市西城区西直门成铭大厦 C 座 807 室

奥莱利技术咨询（北京）有限公司

O’Reilly 为本书维护了一个网页，列出了勘误表、范例以及任何其他信息。可以通过以下地址访问：

http://www.oreilly.com/catalog/9780596520830

要询问技术问题或提出建议，请发邮件至：

bookquestions@oreilly.com

info@mail.oreilly.com.cn

更多关于 O’Reilly 的图书、会议信息、资源中心以及 O’Reilly 网络，请访问以下网页：

http://www.oreilly.com

http://www.oreilly.com.cn

使用示例代码

本书是为了帮助你完成工作。一般情况下，你可以在程序和文档中使用本书的代码。如果你使用本书的重要代码，不必联系我们获取许可。例如，使用本书中几大块代码写自己的程序不需要获得许可，但是如果要将 O’Reilly 书籍中的用例制作成光盘出售或发布，则必须获得许可。引用本书内容或范例解决其他问题不需要获得许可，但是如果想在你的产品文档中包含本书中一些重要的示例代码，那么也需要得到许可。

如果引用了本书内容，那么我们很感激你标明出处，但并不做要求。出处通常包括标题、作者、出版商以及 ISBN。例如，“Learning SQL, Second Edition, by Alan Beaulieu. Copyright 2009 O’Reilly Media, Inc., 978-0-596-52083-0.”。

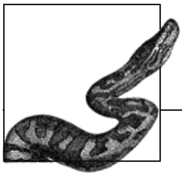
如果你并不清楚使用本书示例代码是否侵权，请随时与我们联系：

permissions@oreilly.com

致谢

首先我想感谢编辑 Mary Treseler，因为她的帮助使第 2 版成为现实，其次，再次感谢 Kevin Kline、Roy Owens、Richard Sonen 和 Matthew Russell，是他们在圣诞节和新年时还在审查本书。我还想感谢第 1 版的一些读者，他们提出了很多问题、建议和勘误。最后，感谢我的妻子 Nancy、女儿 Michelle 和 Nicole，是她们给了我很多鼓励和启发。

目录



第 1 章 背景知识

在我们开始学习本书的内容时，先了解一些数据库方面的基本概念及计算机数据存储和检索的发展史是十分有益的。

1.1 数据库简介

“数据库”是指一组相关信息的集合。例如，电话簿就可以被视为包含某地区所有居民的姓名、电话号码、地址等信息的数据库。尽管电话簿可能是一个最为普及和常用的数据库，但它仍有不少缺点，比如：

- 查找某人的电话号码相当费时，特别是在电话簿包含了海量条目时；
- 电话簿只是根据姓名来索引，因此对于根据特定地址查找居民姓名就无能为力了；
- 当电话簿被打印后，随着该地区居民的流动、更改电话号码或住址等行为不断发生，电话簿上的信息也变得越来越不准确。

电话簿的这些缺陷同样存在于任何人工编制的数据存储系统，比如存放在档案柜的病历等。由于这些纸质数据库不方便，因此最早的计算机应用之一就是开发数据库系统，即通过计算机来存储和检索数据的机制。因为数据库系统通过电子而不是纸质方式来存储数据，所以它可以更快速地检索数据、以多种方式索引数据以及为其用户群提供最新的信息。

早期的数据库系统将被管理的数据存储在磁带上。一般情况下磁带的数量比磁带机要多得多，因此在请求数据时需要技术人员手动装卸磁带。同时由于那个时代的计算机内存很小，通常对同一数据的并发请求必须多次读取磁带，降低了使用效率。因此尽管这些数据库系统相对于纸质数据库有了显著的进步，但与今天的数据库技术相比仍有相当遥远的差距。（现代数据库系统能够利用海量快速的磁盘驱动器来管理太字节级的数据，在高速内存中存放数十吉字节的数据。）

1.1.1 非关系数据库

提示

本小节讨论早期非关系数据库的背景信息，如果读者急于学习 SQL，可以跳过下面几页，直接阅读下一章节。

在计算机数据库发展的前几十年里，数据以各种不同的方式存储并展现给用户。例如，在层次数据库系统中，以一个或多个树形结构来表示数据。图 1-1 显示了以树形结构表示的 George Blake 和 Sue Smith 的银行账户数据。

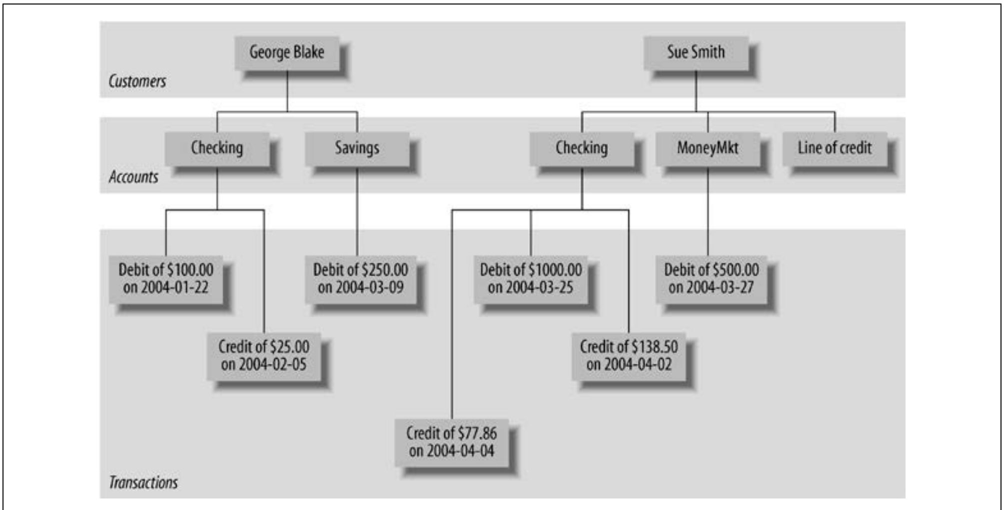


图 1-1　账户数据的层次视图

George 和 Sue 的数据树都包含了各自的账户以及交易信息。层次数据库系统提供了定位客户信息树的工具，并能够遍历此树找到所需要的账户或交易数据。树中的每个节点都具有 0 个或 1 个父节点，以及 0 个、1 个或多个子节点。这种设置被称为单根层次结构（single-parent hierarchy）。

另一种管理数据的方式是网状数据库系统，它表现为多个记录集合，集合之间通过链接来定义不同记录间的关系。图 1-2 显示了使用此系统表示的 George 和 Sue 的账户信息。

在此系统中，为了查找 Sue 的 MoneyMkt 账户上的交易信息，需要执行下面的步骤：

1．查找 Sue Smith 的客户记录；

2．通过链接从 Sue Smith 的客户记录找到其账户列表；

3．遍历账户列表直至找到 MoneyMkt 账户；

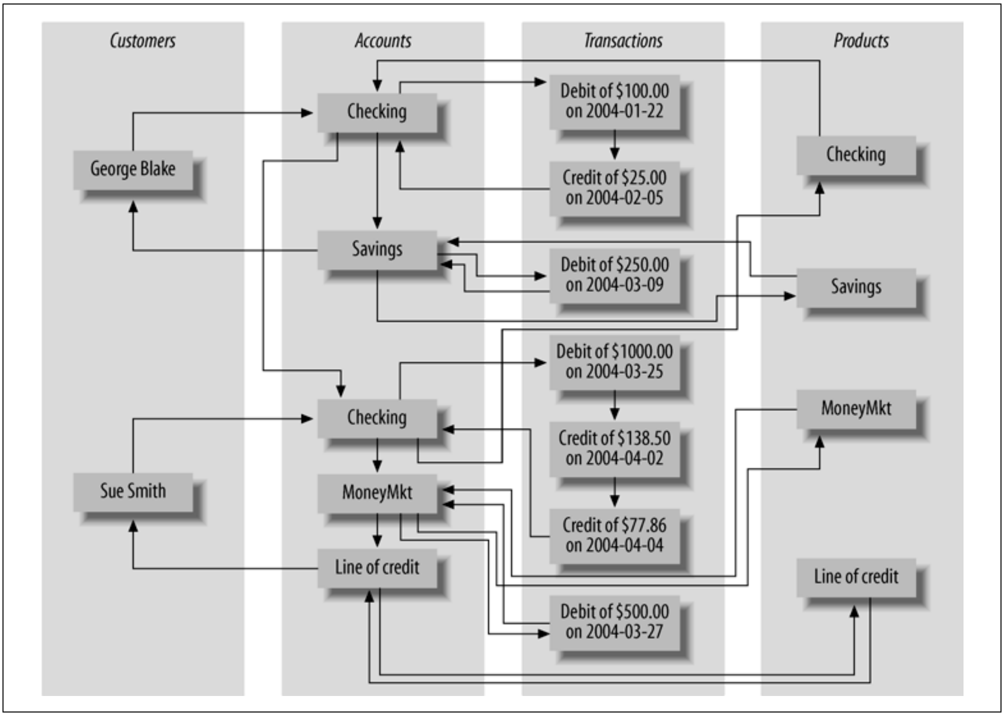


图 1-2　账户数据的网状视图

4．　通过链接从 MoneyMkt 账户找到其交易列表。

网状数据库系统的有趣特性体现在图 1-2 中最右侧的 products 记录。注意每个 product 记录（Checking、Savings 等）都指向一个 account 记录列表，以指定这些账户记录的产品类型。因此 account 记录可以通过多个入口进行访问（customer 记录或 product 记录），这使得网状数据库具有多根层次的特点。

层次和网状数据库仍然存在，尽管主要在大型机领域中使用。此外，层次数据库系统与可扩展标记语言（XML）相结合，在目录服务方面获得了新的应用，比如 Microsoft 公司的 Active Directory 和 Red Hat 的 Directory Server。然而，从 20 世纪 70 年代开始，一种新的表示数据的方式逐步扎根并获得发展，这种方式更为严谨，且易于理解和实现。

1.1.2　关系模型

1970 年，IBM 研究院的 E.F.Codd 博士发表了一篇名为“大型共享数据银行的数据关系模型”的论文，提出使用表集合来表示数据，但相关条目之间并不使用指针来导航，而是借助冗余数据来链接不同表中的记录。图 1-3 显示了此方法所表示的 George 和 Sue 的账户信息。

Customer

cust_id	fname	lname
1	George	Blake
2	Sue	Smith

Account

account_id	product_cd	cust_id	balance
103	CHK	1	$75.00
104	SAV	1	$250.00
105	CHK	2	$783.64
106	MM	2	$500.00
107	LOC	2	0

Product

product_cd	name
CHK	Checking
SAV	Savings
MM	Money market
LOC	Line of credit

Transaction

txn_id	txn_type_cd	account_id	amount	date
978	DBT	103	$100.00	2004-01-22
979	CDT	103	$25.00	2004-02-05
980	DBT	104	$250.00	2004-03-09
981	DBT	105	$1000.00	2004-03-25
982	CDT	105	$138.50	2004-04-02
983	CDT	105	$77.86	2004-04-04
984	DBT	106	$500.00	2004-03-27

图 1-3　账户数据的关系视图

图 1-3 中包含了 4 个记录表：Customer、Product、Account 和 Transaction。首先查看图中顶部的 Customer 表，其中具有 3 列：cust_id（其中包含客户的 ID 号）、fname（其中包含客户的名字）和 lname（其中包含客户的姓氏）。Customer 表中包含了两个记录行，分别为 Georage Blake 和 Sue Smith 的数据。在不同的数据库管理服务器中，记录表可包含的最大列数是有差异的，但通常这个数目足够大而不需要为此担心（比如 Microsoft 的 SQL Server 允许每张表可以最多具有 1024 列）。表中数据行的数目通常只是受到物理设备（磁盘空间大小）和可维护性（在表中记录数在到达多大规模之后仍能保持易用性）的限制，而数据库管理服务器一般不对此进行限制。

关系数据库中的每张表都包含一项作为每行唯一标识的信息（主键），它与其他信息一起构成了对条目的完整描述。对于 Customer 表，cust_id 列为每个顾客保存了不同的编号。例如，George Blake 可以由顾客 ID#1 来唯一标识，其他顾客则永远不会被赋予此标识符。因此在 Customer 表中，不再需要其他信息来定位 George Blake 的数据。

提示

每种数据库服务器都提供用于产生一组作为主键值的唯一数字的机制，因此用户不需要为哪些数字已被赋予为主键而操心。

当然也可以选择联合使用 fname 和 lname 两列作为主键（包含两列或多列的主键通常被称为复合主键），实际上，在银行账户中很可能会出现两个或多个人具有完全相同的姓氏和名字。因此，选择 cust_id 列专门用于 Customer 表的主键是更合适的做法。

提示

在本例中，选择 fname/lname 作为主键，称之为自然主键；使用 cust_id 作为主键，则称为逻辑主键。使用哪一种主键更好一直是悬而未决的热门讨论问题，但在本例中的选择是显而易见的，因为人们的姓名可能发生改变（比如某人结婚后使用其配偶的姓氏），而主键列在被赋值后是绝不允许被修改的。

一些表中还包含了导航到其他表的信息，即前文提到的“冗余数据”。举例来说，Account 表中的 cust_id 列，它包含了使用该账户的顾客的唯一性标识，而 product_cd 列则包含了该账户所关联产品的唯一性标识。这些列被称为外键，用于作为账户信息网络结构中的各实体之间的连线。如果需要查询某个账户所有者的相关信息，则需要获取 cust_id 列的值，并使用它在 Customer 表中查找相应的行（在关系数据库术语中，此过程被称为连接（join），在第 3 章基本查询对此进行了介绍，并在第 5 章和第 10 章进行了更深入的讨论）。

也许多次存储同样的信息是一种浪费的做法，但是某些情况下使用冗余数据能够更清晰地体现关系模型。例如，在 Account 表中包含一个该账户所有者的唯一标识符是合适的，如果在 Account 表中再增加顾客的 fname/lname 列就不太恰当了，这会使数据库中的数据不再可靠。因为放置该数据（顾客的姓/名）的地方应当是 Customer 表，并且该表中只有 cust_id 列才适合在其他表中被引用。此外，在一列中包含多条信息也是不合适的，比如使用 name 列同时包含顾客的姓和名，或者使用 address 列包含街道、城市、省以及邮政编码等信息。数据库设计精化过程的主要目标就是保证每条独立的信息只存放在一个地方（外键除外），称为规范化。

返回到图 1-3 中的 4 个表，或许你会疑惑如何使用这些表来查找 George Blake 在他的 checking 账户上的交易信息。首先，在 Customer 表中找到 George Blake 的唯一性主键。然后，在 Account 表中找到 cust_id 列等于 George 的唯一性标识符，并通过 Product_cd 匹配 Product 表中 name 列为“Checking”的那些行。最后，通过匹配 Account 表的唯一性标识 account_id 列来定位 Transaction 表中相对应的行。这些看起来有些复杂，但你很快就会发现，在 SQL 语言中，使用一个命令就足以完成这些任务了。

1.1.3 一些术语

在前面已经介绍了一些新的术语，下面介绍一些正式的定义，表 1-1 显示了本书余下部分所使用术语的定义。

表 1-1　术语和定义

术语	定　　义
实体	数据库用户所关注的对象，如顾客、部门、地理位置等
列	存储在表中的独立数据片段
行	所有列的一个集合，完整地描述了一个实体或实体上的某个行为，也称之为记录
表	行的集合，既可以保存在内存中（未持久化），也可以保存在存储设备中（已持久化）
结果集	未持久化表的另一个名字，一般为 SQL 查询的结果
主键	用于唯一标识表中每个行的一个或多个列
外键	一个或多个用于识别其他表中某一行的列

1.2　什么是 SQL

根据 Codd 对关系模型的定义，他提出一种名为 DSL/Alpha 的语言，用于操控关系表的数据。在 Codd 的论文发表后不久，IBM 建立了一个研究小组来根据他的想法构建原型。该小组创建了一个 DSL/Alpha 的简化版本，即 SQUARE，然后通过对 SQUARE 的改进，将之发展为 SEQUEL 语言，并最终命名为 SQL。

今天 SQL 已经发展到了中年期（唉，就像作者一样），在这期间它经历了大量修改。在 20 世纪 80 年代中期，美国国家标准组织（ANSI）开始制定 SQL 语言的第一个标准，并于 1986 年发布。其后不断对其改进，并在 1989 年、1992 年、1999 年、2003 年和 2006 年发布了一系列 SQL 标准的新版本。通过对语言核心的改良，新的特性被陆续加入到 SQL 语言中，以吸收面向对象等其他功能。最后一个标准版本，SQL 2006 则聚焦于 SQL 和 XML 的集成，并定义了 XQuery 语言以用于在 XML 文档中查询数据。

SQL 与关系模型的关系密切，因为 SQL 查询的结果也可以视为一张表（在程序上下文中称之为结果集）。因此，可以在关系数据库中简单地创建一个固定表，用于存放查询的结果集。同样地，SQL 查询也可以使用固定表或其他查询的结果集作为其输入（在第 9 章中将会讲述其细节）。

最后需要注意的一点是：SQL 并不是任何短语的缩写（尽管许多人坚持认为它代表结构化查询语言（Structured Query Language））。当提到此语言时，可以使用独立的字母（S.Q.L）或使用单词 sequel。

1.2.1　SQL 语句的分类

在本书中，将分别讨论 SQL 语言的几个独立模块，即 SQL 方案（schema）语句，用于定义存储于数据库中的数据结构；SQL 数据语句，用于操作 SQL 方案语句所定义的数据结构；以及 SQL 事务语句，用于开始、结束或回滚事务（将在第 12 章中介绍）。

例如，在数据库中创建新表时，需要使用 SQL 方案语句 create table，而在新表中产生数据则需要 SQL 数据语句 insert。

下面给出这些语句的具体例子，用于创建 corporation 表的 SQL 方案语句如下：

```
CREATE TABLE corporation
 (corp_id SMALLINT,
  name VARCHAR(30),
  CONSTRAINT pk_corporation PRIMARY KEY (corp_id)
 );
```

该语句创建的表包括两列：corp_id 和 name。其中，corp_id 列被设置为表的主键。在第 2 章中，将会介绍该语句的细节，比如 MySQL 中所提供的各种数据类型。下面给出的 SQL 数据语句将向 corporation 表中插入一行关于 Acme Paper Corporation 的数据：

```
INSERT INTO corporation (corp_id, name)
VALUES (27,'Acme Paper Corporation');
```

该语句向 corporation 表中添加了一行数据，其中 corp_id 列的值为 27，而 name 列的值是 Acme Paper Corporation。

最后，给出一条简单的 select 语句，以获取刚才创建的数据：

```
mysql> SELECT name
    -> FROM corporation
    -> WHERE corp_id = 27;
+------------------------+
| name                   |
+------------------------+
| Acme Paper Corporation |
+------------------------+
```

通过 SQL 方案语句所创建的所有数据库元素都被存储在一个特殊的表集合，即数据字典中。这些“关于数据库的数据”一般被称为“元数据”，本书第 15 章将对此进行详细介绍。与用户所创建的表一样，数据字典表也可以通过 select 语句查询，从而允许在运行时刻查看数据库中的当前数据结构。例如，用户需要编写显示上月新增账户的报表，那么既可以在报表中对 account 表的各个列名进行硬编码，也可以通过查询数据字典以获取当前的列集合并在每次运行时动态地创建报表。

本书中的大部分篇幅将聚焦于 SQL 语言中的数据相关部分，包括 select、update、insert 和 delete 命令。SQL 方案语句将在第 2 章中说明，并且该章所创建的示例数据库将在全书中使用。一般来说，不需要对 SQL 方案语句的语法进行太多论述，而对于 SQL 数据语句，尽管只有寥寥几条，但其中包含了大量值得仔细研究的内容。因此，尽管我尽量介绍更多的 SQL 方案语句,但本书的大多数章节还是把重点放在 SQL 数据语句上。

1.2.2　SQL：非过程化语句

如果读者有过编程语言的使用经验，可能习惯于定义变量或数据结构、使用条件逻辑

（即 if-then-else）和循环结构（即 do-while-end），并将程序代码分成可复用的小片段（如对象、函数、过程等）。这些代码经过编译后执行，其执行结果精确地（也并不是总是精确）符合编程的预期。无论是使用 Java、C#、C、Visual Basic 还是其他过程化语言，都可以完全控制程序的行为。

提示

过程化语言对所期望的结果和产生这些结果的执行机制或过程都进行了定义。非过程化语言同样定义了期望结果，但将产生结果的过程留给外部代理来定义。

使用 SQL 意味着必须放弃对过程的控制，因为 SQL 语句只定义必要的输入和输出，而执行语句的方式则交由数据库引擎的一个组件，即优化器（optimizer）处理。优化器的工作包括查看 SQL 语句并考虑该表的配置信息以及有无索引等，以确定最具效率的执行路径（当然，并不总是最有效率）。大多数数据库引擎允许通过指定优化器选项来影响优化器的决策，比如建议使用特定的索引等。而大多数 SQL 的用户并不需要考虑这个复杂的层面，而是将之交给数据库管理员或性能调优专家来处理。

因此单独使用 SQL 并不能开发完整的应用，除了编写简单的脚本来处理某些数据，一般需要将 SQL 与编程语言相集成。一些数据库厂商已经为用户考虑了这些，如 Oracle 的 PL/SQL 语言，MySQL 的存储过程语言，以及 Microsoft 的 Transact-SQL 语言。在这些语言中，SQL 数据语言是其语法的一部分，以准确无误地将数据库查询与过程化命令集成到一起。如果使用非数据库指定的语言，如 Java 等，则需要使用工具集/API 以在代码中执行 SQL 语句。有些工具集由数据库厂商提供，其他的则由第三方厂商或开源代码提供者所创建。表 1-2 显示了将 SQL 集成到特定语言的可用选项。

表 1-2　SQL 集成工具集

语言	工　具　集
Java	JDBC (Java Database Connectivity; JavaSoft)
C++	Rogue Wave SourcePro DB (third-party tool to connect to Oracle, SQL Server, MySQL, Informix, DB2, Sybase, and PostgreSQL databases)
C/C++	Pro*C (Oracle), MySQL C API (open source), and DB2 Call Level Interface (IBM)
C#	ADO.NET (Microsoft)
Perl	Perl DBI
Python	Python DB
Visual Basic	ADO.NET (Microsoft)

如果用户仅仅需要执行交互式的命令，那么每种数据库开发商都提供了至少一个简单

的命令行工具，用于向数据库引擎提交 SQL 命令。大多数开发商都提供了图形化的工具，其中包含显示 SQL 命令的窗口以及另一个显示 SQL 命令执行结果的窗口。因为本书中的例子都将在 MySQL 数据库中运行，所以本书使用 mysql 命令行工具。该工具属于 MySQL 安装文件的一部分，并用于运行示例和格式化的结果。

1.2.3 SQL 示例

在本章前面,我说过要演示返回 George Blake 的 checking 账户上所有交易的 SQL 语句，下面就兑现这个承诺，语句和查询结果如下：

```
SELECT t.txn_id, t.txn_type_cd, t.txn_date, t.amount
FROM individual i
  INNER JOIN account a ON i.cust_id = a.cust_id
  INNER JOIN product p ON p.product_cd = a.product_cd
  INNER JOIN transaction t ON t.account_id = a.account_id
WHERE i.fname = 'George' AND i.lname = 'Blake'
  AND p.name = 'checking account';
+--------+-------------+---------------------+--------+
| txn_id | txn_type_cd | txn_date            | amount |
+--------+-------------+---------------------+--------+
|    11  | DBT         | 2008-01-05 00:00:00 | 100.00 |
+--------+-------------+---------------------+--------+
1 row in set (0.00 sec)
```

此处仅对此语句进行简单的分析：该查询查找满足下面两个条件的行，即在 individual 表中姓名为 George Blake 的行，以及在 product 表中的账户名为 checking account 的行，并通过 account 表将它们关联起来，然后返回 transaction 表中所有提交到该账户上的交易信息内容，并分 4 列显示。如果刚好知道 George Blake 的客户 ID 是 8 并且 checking 账户的指定代码为“CHK”，就可以简单地根据客户 ID 找到 George Blake 在 account 表中的 checking 账户，并使用账户 ID 来查找相关的交易：

```
SELECT t.txn_id, t.txn_type_cd, t.txn_date, t.amount
FROM account a
  INNER JOIN transaction t ON t.account_id = a.account_id
WHERE a.cust_id = 8 AND a.product_cd = 'CHK';
```

在下面的各章节里，将会覆盖到这些查询的所有概念（并且会涉及得很多），但在这里至少需要展示一下它们的大致结构。

前面的查询包含了 3 个不同的字句，包括 select、from 和 where。几乎所有的查询都至少会包含这 3 个子句，当然还有其他几个子句可用于更特定的查询目标。下面展示了这 3 个子句所起的角色：

```
SELECT /* 1个或多个事物*/ ...
FROM /* 1个或多个地点*/ ...
WHERE /* 1个或多个条件*/ ...
```

提示

大多数的 SQL 实现都将/*和*/标记之间的文本视为注释。

当用户构造查询时，首先需要确定查找的是哪一个或哪些表，并将它们加入 from 子句中，然后在 where 子句中增加查询条件以过滤掉并不感兴趣的数据。最后，需要确定从各个表中所应提取的列，并将之增加到 select 子句中。下面给出一个简单的例子，以展示如何找到所有姓为“Smith”的客户：

```
SELECT cust_id, fname
FROM individual
WHERE lname = 'Smith';
```

该查询搜索 individual 表，以找到所有 lname 列匹配字符串'Smith'的行，并返回这些行中的 cust_id 和 fname 列。

除了查询数据库，还需要在数据库中建立和修改数据，下面举出一个简单的例子，以说明如何在 product 表中插入新行：

```
INSERT INTO product (product_cd, name)
VALUES ('CD', 'Certificate of Depysit')
```

糟糕，这里将“Deposit”拼错了，不过没有关系，可以使用 update 语句来修复这个错误：

```
UPDATE product
SET name = 'Certificate of Deposit'
WHERE product_cd = 'CD';
```

注意，与 select 语句一样，update 语句也包含了 where 子句，这是因为 update 语句也要识别所需修改的行。在本例中，只需要将要修改的行指定为 product_cd 列与字符串'CD'相匹配的那些行即可。由于 product_cd 列是 product 表的主键，因此可以预计 update 语句会精确地修改某一行（或零行，如果表中该值不存在）。在任何时刻执行 SQL 数据语句，都会收到一个来自数据库引擎的反馈，以显示该语句所影响的行数。如果使用交互式工具，比如上文提到的 mysql 命令行工具，那么可以接收到下面几种操作所影响行数的反馈：

- select 语句的返回行数；
- insert 语句创建的行数；
- update 语句修改的行数；
- delete 语句所删除的行数。

可以使用过程化语言，并结合上文提到的工具集来调用 SQL 语句。工具集通常包含了能够获取 SQL 数据语句执行信息的调用。一般来说，好的做法应当是检查这个信息以

确信语句执行并没有超出预料（比如忘记为 delete 语句增加 where 子句，从而删除了表中的所有行）。

1.3 什么是 MySQL

商业级关系数据库已经存在 20 多年了，几种最成熟和流行的商业产品包括：

- 甲骨文公司的 Oracle Database；
- Microsoft 公司的 SQL Server；
- IBM 公司的 DB2 Universal Database；
- Sybase 公司的 Sybase Adaptive Server。

这些数据库服务器的功能十分类似，尽管它们中的一些擅长处理大容量和高流量的数据库，而另一些对于处理对象、大文件或 XML 文档等更为适合，所有这些服务器都遵从了最新的 ANSI SQL 标准。这是一件好事，本书将演示如何编写标准的 SQL 语句，以便无须修改（或极少量的修改）就能够在这些平台中运行。

在最近 5 年里，除了商业级数据库服务器，开源社区也为创建商业数据库产品的可替代品而努力，其中两个最常用的开源数据库服务器为 PostgreSQL 和 MySQL。MySQL 的主页上（http://www.mysql.com）声称其已经拥有超过 1000 万次的安装，它的服务器版式是免费使用的，并且该服务器软件的下载和安装都非常简单。出于这些理由，本书的所有示例都将在 MySQL（6.0 版）上运行，并使用 mysql 命令行工具格式化查询结果。即使你已经使用了另一种数据库且从未打算使用 MySQL，本书还是建议安装 MySQL 服务器的最新版本，并载入书中示例所包含的 SQL 方案和数据语句。

不过，读者还需要牢记下面的说明：

本书并不是一本 MySQL 的 SQL 实现教程。

事实上，本书原意是希望教授如何设计 SQL 语句并使之无需修改地运行在 MySQL 上，并能在无需或仅需要极少量修改的情况下，运行在 Oracle Database、Sybase Adaptive Server 和 SQL Server 上。

为了使本书中的代码尽量保持数据库平台版本独立性，作者不得不克制对 MySQL SQL 语言一些有趣特性的介绍，因为这些特性在其他数据库实现上不能被完成。作为补充，附录 B 覆盖了其中一些特性，以帮助那些准备持续使用 MySQL 的读者。

1.4 内容前瞻

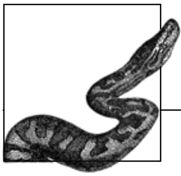
接下来 4 章的主要目标是简介 SQL 数据语句，重点放在 select 语句的 3 个主要子句上。

此外还提供了许多银行业务方面的实例（在下一章中介绍），本书中所有的示例都围绕它们展开。这是因为使用同一个已熟悉的数据库作为例子，将会更容易地掌握问题的核心，而不是每次都需要了解所使用的表。如果读者对总是使用同样的表集合感到厌倦，那么可以在示例数据库中自由增加新表，或者干脆建立自己的试验数据库。

在帮助读者牢固掌握了基础知识后，剩余的章节将会深入讨论更多的概念，它们大多是相互独立的。因此，读者可以根据自己的疑问，自由地向前或向后浏览某个章节。当你完整阅读本书并使用过所有的示例后，你就已经在通往 SQL 专家的路上迈出了坚实的第一步。

在本章之外，如果读者还需要了解更多关系数据库、计算机数据库系统的历史以及 SQL 语言方面的知识，可以参考下面列出的一些资源：

- C.J. Date’s Database in Depth: Relational Theory for Practitioners (O’Reilly)
- C.J. Date’s *An Introduction to Database Systems*, Eighth Edition (Addison-Wesley)
- C.J. Date’s *The Database Relational Model: A Retrospective Review and Analysis: A Historical Account and Assessment of E. F. Codd’s Contribution to the Field of Database Technology* (Addison-Wesley)
- http://en.wikipedia.org/wiki/Database_management_system
- http://www.mcjones.org/System_R/



第 2 章

创建和使用数据库

本章提供的内容包括如何创建数据库、如何创建表，并且相关的数据将会作为全书的用例。此外还介绍了各种数据类型以及如何在建表时使用它们。因为本书中的示例都运行在 MySQL 数据库上，因此本章比较偏向使用一些 MySQL 的特性和记号，但大多数概念对其他数据库服务器也是适合的。

2.1 创建 MySQL 数据库

如果读者已经有了可以使用的 MySQL 数据库服务器，那么可以跳过下面的安装指南，直接阅读表 2-1。不过需要注意，本书使用的 MySQL 为 6.0 及以上版本，如果你使用的是以前的版本，就需要将服务器升级或者安装另一个服务器。

表 2-1　创建示例数据库

步　骤	描　述	行　为
1	从开始菜单打开运行对话框	打开开始菜单并选择运行
2	调出命令窗口	在对话框中键入 cmd 并单击确定按钮
3	用 root 用户登录 MySQL	mysql -u root -p
4	创建示例数据库	create database bank;
5	创建用户 lrngsql，并赋予 bank 数据库的权限	grant all privileges on bank.* to 'lrngsql'@'localhost' identified by 'xyz';
6	退出 mysql 工具包	quit;
7	使用 lrngsql 用户登录 MySQL	mysql -u lrngsql -p;
8	关联 bank 数据库	use bank;

下面的操作指南显示了在 Windows 操作系统上安装 MySQL 6.0 服务器版所需要的最少

步骤。

1． 访问 MySQL 数据库服务器的下载页面：http://dev.mysql.com/downloads。其中所要下载的 6.0 版本完整的 URL 为 http://dev.mysql.com/downloads/mysql/6.0.html。

2． 下载 Windows (x86)下的基础压缩包，其中包含了最常用的工具。

3． 当弹出“Do you want to run or save this file?”确认窗口后，单击下一步按钮。

4． 等待出现 MySQL 服务器 6.0 版的安装向导窗口，然后单击下一步按钮。

5． 选择典型安装，单击下一步按钮。

6． 单击安装按钮。

7． 出现 MySQL 企业版的安装窗口，连续单击两次下一步按钮。

8． 当安装完成后，选中 Configure the MySQL Server now 选项，然后单击完成按钮，以载入配置向导界面。

9． 当配置向导界面被载入后，激活标准配置选项按钮，然后同时选中 Install as Windows Service 和 Include Bin Directory in Windows Path 复选框，单击下一步按钮。

10．选择修改安全设置复选框，并输入 root 用户的密码（记住所输入的密码，因为你马上就要用到它！），然后单击下一步按钮。

11．单击执行按钮。

到此，如果一切顺利，MySQL 服务器就会被成功安装并运行，否则建议卸载服务并阅读“MySQL 在 Windows 下的安装疑难解答”指南（可以在 http://dev.mysql.com/doc/refman/6.0/en/windows-troubleshooting.html 上找到它）。

警告

如果在安装 6.0 版本之前卸载了旧版本的 MySQL，那么可能还需要更进一步清理（比如清除一些旧的注册表条目），以便能够成功获取设置向导。

接下来需要打开 Windows 命令窗口，调用 mysql 工具，并创建数据库和用户。表 2-1 列出了必需的步骤。在第五步中，也可以为 lrngsql 用户选择与“xyz”不同的密码（但不要忘了所输入的密码哦！）。

现在已经准备好了 MySQL 服务器、示例数据库以及数据库用户，下面只剩下创建数据库表并产生示例数据的工作了。可以在 http://examples.oreilly.com/learningsql/下载相关脚本，并使用 mysql 工具运行。假设将该脚本文件存放为 c:\temp\LearningSQLExample.sql，接下来需要完成以下两项任务：

1． 如果已经从 mysql 工具包中退出，那么需要重复表 2-1 中的第 7 和第 8 步；

2． 输入源文件路径 c:\temp\LearningSQLExample.sql，并按 Enter 键。

如此就顺利创建了本书所有例子所使用的数据库，并产生了相关数据。

2.2 使用 mysql 命令行工具

在调用 mysql 命令行工具时，可以同时指定用户名和所要使用的数据库，如下所示：

```
mysql -u lrngsql -p bank
```

这可以省去每次启动工具时都要输入 use bank 命令的麻烦。然后提示需要输入密码，输入完成后会立即出现 mysql>标记，在该标记的后面可以运行 SQL 语句并查看输出结果。例如，需要知道当前日期和时间，可以输入下面的查询：

```
mysql> SELECT now();
+---------------------+
| now()               |
+---------------------+
| 2008-02-19 16:48:46 |
+---------------------+
1 row in set (0.01 sec)
```

now()是内建的 MySQL 函数，它返回当前日期与时间。如上所示，mysql 命令行工具使用+、-和|等符号将查询结果格式化输出在矩形框中。显示完查询结果（本例的结果只有 1 行）之后，mysql 命令行工具还显示返回的行数，以及 SQL 语句执行的时间。

缺失的子句

某些数据库服务器规定查询语句中必须包含 from 子句，并在其中至少指明一个表名，比如广泛使用的 Orcale 数据库。这时如果是仅仅需要调用一个函数，Oracle 为此提供了一个特殊的表 dual，该表只包含一个名为 dummy 的列，并且只会有一个数据行。为了与 Oracle 数据库保持兼容，MySQL 也提供了 dual 表，因此前面的关于当期日期时间的查询可以使用下面的语句：

```
mysql> SELECT now()
       FROM dual;
+---------------------+
| now()               |
+---------------------+
| 2005-05-06 16:48:46 |
+---------------------+
1 row in set (0.01 sec)
```

如果没有使用 Oracle，并且不需要与之兼容，可以省略 dual 表，只使用不带 from 子句的 select 语句。

使用完 mysql 命令行语句后，可以简单地输入 quit 或 exit，以返回 Windows shell。

2.3 MySQL 数据类型

一般来说，所有流行的数据库都可以存储同样的数据类型，比如字符串、日期和数字等，但对于一些特殊的数据类型，比如 XML 文档或大的文本及二进制文档，各种数据库之间存在着较为明显的差异。本书仅针对 SQL 语言进行介绍，通常用到的数据列 98%都是简单数据类型，因此本书只涉及字符型、数值型和日期型。

2.3.1 字符型数据

字符型数据可以使用定长或变长的字符串来实现，其不同点在于固定长度的字符串使用空格向右填充，以保证占用同样的字节数；变长字符串不需要向右填充，并且所有字节数可变。当定义一个字符列时，必须指定该列所能存放字符串的最大长度。例如，需要存储最大长度不超过 20 个字符的字符串，可以使用下面的定义方式：

```
char(20)    /* fixed-length */
varchar(20) /* variable-length */
```

char 列可以设置的最大长度为 255 个字节，而 varchar 列最多可以存储 65535 个字节。如果需要存储更长的字符串（比如电子邮件、XML 文档等），则需要使用文本类型（mediumtext 和 longtext），后面将对此进行介绍。一般情况下，使用 char 类型来存储同样长度的字符串，比如州名的简写，以及使用 varchar 类型来存储变长字符串。在所有主流数据库中，char 和 varchar 的使用方式都是类似的。

提示

Oracle 数据库对 varchar 的使用是个特例，使用 varchar2 类型表示变长字符串列。

字符集

对于拉丁系语言，比如英语，包含了一系列字母，其中每个字母只需要 1 个字节来存储。其他一些语言（如日语和韩语）则包含了大量字符，每个字符的存储需要多个字节，因此这类字符集被称为多字符集。

MySQL 可以使用各种字符集来存储数据，包括单字符集和多字符集。可以使用 show 命令来查看服务器所支持的字符集，如下所示。

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------+-------------------+--------+
| Charset  | Description           | Default collation | Maxlen |
+----------+-----------------------+-------------------+--------+
| big5     | Big5 Traditional Chinese| big5_chinese_ci |     2  |
| dec8     | DEC West European     | dec8_swedish_ci   |     1  |
| cp850    | DOS West European     | cp850_general_ci  |     1  |
```

```
| hp8      | HP West European            | hp8_english_ci      |  1 |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |  1 |
| latin1   | cp1252 West European        | latin1_swedish_ci   |  1 |
| latin2   | ISO 8859-2 Central European | latin2_general_ci   |  1 |
| swe7     | 7bit Swedish                | swe7_swedish_ci     |  1 |
| ascii    | US ASCII                    | ascii_general_ci    |  1 |
| ujis     | EUC-JP Japanese             | ujis_japanese_ci    |  3 |
| sjis     | Shift-JIS Japanese          | sjis_japanese_ci    |  2 |
| hebrew   | ISO 8859-8 Hebrew           | hebrew_general_ci   |  1 |
| tis620   | TIS620 Thai                 | tis620_thai_ci      |  1 |
| euckr    | EUC-KR Korean               | euckr_korean_ci     |  2 |
| koi8u    | KOI8-U Ukrainian            | koi8u_general_ci    |  1 |
| gb2312   | GB2312 Simplified Chinese   | gb2312_chinese_ci   |  2 |
| greek    | ISO 8859-7 Greek            | greek_general_ci    |  1 |
| cp1250   | Windows Central European    | cp1250_general_ci   |  1 |
| gbk      | GBK Simplified Chinese      | gbk_chinese_ci      |  2 |
| latin5   | ISO 8859-9 Turkish          | latin5_turkish_ci   |  1 |
| armscii8 | ARMSCII-8 Armenian          | armscii8_general_ci|  1 |
| utf8     | UTF-8 Unicode               | utf8_general_ci     |  3 |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |  2 |
| cp866    | DOS Russian                 | cp866_general_ci    |  1 |
| keybcs2  | DOS Kamenicky Czech-Slovak  | keybcs2_general_ci  |  1 |
| macce    | Mac Central European        | macce_general_ci    |  1 |
| macroman | Mac West European           | macroman_general_ci|  1 |
| cp852    | DOS Central European        | cp852_general_ci    |  1 |
| latin7   | ISO 8859-13 Baltic          | latin7_general_ci   |  1 |
| cp1251   | Windows Cyrillic            | cp1251_general_ci   |  1 |
| cp1256   | Windows Arabic              | cp1256_general_ci   |  1 |
| cp1257   | Windows Baltic              | cp1257_general_ci   |  1 |
| binary   | Binary pseudo charset       | binary              |  1 |
| geostd8  | GEOSTD8 Georgian            | geostd8_general_ci  |  1 |
| cp932    | SJIS for Windows Japanese   | cp932_japanese_ci   |  2 |
| eucjpms  | UJIS for Windows Japanese   | eucjpms_japanese_ci|  3 |
+---------+-----------------------------+--------------------+----+
36 rows in set (0.11 sec)
```

如果其中的第 4 列 maxlen 大于 1，那么该字符集为多字符集。

在安装 MySQL 服务器时，latin1 字符集将会被自动选择为默认字符集。当然，还可以为数据库中的每个字符列选择不同的字符集，甚至可以在同一个数据表内存储不同的字符集数据。为数据列指定非默认的字符集，只需要在类型定义后加上系统支持的字符集名称，例如：

```
varchar(20) character set utf8
```

在 MySQL 中，还可以改变整个数据库的默认字符集：

```
create database foreign_sales character set utf8;
```

尽管本书中对字符集的介绍就到此为止了，但实际上关于国际化的主题还包含了更为

广泛的内容。如果你需要处理多种非母语的字符集，可以参考 Andy Deitsch 和 David Czarnecki’s *Java Internationalization* 或 Richard Gillam 的 *Unicode Demystified: A Practical Programmer’s Guide to the Encoding Standard* 等著作。

文本数据

如果需要存储的数据超过 64KB（varchar 列所能容许的上限），就需要使用文本类型。

表 2-2 显示了可用的文本类型及它们的最大长度。

表 2-2　MySQL 文本类型

文 本 类 型	Maximum number of bytes
tinytext	255
text	65 535
mediumtext	16 777 215
longtext	4 294 967 295

当选择使用文本类型时，应注意下列事项。

- 如果被装载到文本列中的数据超出了该类型的最大长度，数据将会被截断。
- 在向文本列装载数据时，不会消除数据的尾部空格。
- 当使用文本列排序或分组时，只会使用前 1024 个字节，当然在需要时可以放宽这个限制。
- 这些不同的文本类型只是针对 MySQL 服务器。SQLServer 对于大的字符型数据只提供 text 类型，而 DB2 和 Oracle 使用的数据类型名称为 clob，即 Character Large Object。
- 如今 MySQL 允许 varchar 列最大容纳 65536 个字节（在 MySQL 4 中仅限制为 255 个字节），这样一般不需要使用 tinytext 或 text 类型了。

如果创建的列用于存储自由格式的数据条目，比如用于存储客户与公司客服部门交互数据的 notes 列，那么一般使用 varchar 就足够了。不过如果需要存储文档，那就可以选择 mediumtext 或 longtext 类型。

提示

Oracle 数据库中，char 列能容纳 2000 个字节，varchar2 列能容纳 4000 个字节，而 SQL Server 中 char 和 varchar 列都能够容纳 8000 个字节。

2.3.2　数值型数据

尽管使用独立的数值数据类型“numeric”似乎更为合理，但实际上存在着好几种不同

的数值数据类型，它们反映了数字应用的不同方式，如下所述。

某列需要指示顾客订单是否已被发送

　　该列类型可以被设为 Boolean，它的值为 0 表示否，为 1 表示是。

交易表中由系统自动生成的主键

　　该列数据由 1 开始，并每次自增 1，可能会达到非常大的数字。

顾客电子购物篮的物品号

　　该列的值应当为正数，从 1 到 200（假设 200 为购物篮所能容纳的最多物品数）。

电路板钻孔机的位置数据

　　高精度的科学和制造业数据往往需要精确到小数点后 8 位。

为了处理这些类型的数据，MySQL 提供几种不同的数值数据类型，最常用的是用于存储所有整数的类型，在这些类型前面还可以加上 unsigned 关键字，以向服务器指明该列存储的数据大于等于 0。表 2-3 显示了 5 种用于存储整数的类型。

表 2-3　MySQL 的整数类型

类　　型	带符号的范围	无符号的范围
tinyint	−128～127	0～255
smallint	−32 768～32 767	0～65 535
mediumint	−8 388 608～8 388 607	0～16 777 215
int	−2 147 483 648～2 147 483 647	0～4 294 967 295
bigint	−9 223 372 036 854 775 808～ 9 223 372 036 854 775 807	0～18 446 744 073 709 551 615

当使用这些整数类型之一创建列时，MySQL 将为存储数据分配合适大小的空间，从 1 个字节（tinyint）到 8 个字节（bigint）不等。因此在选择类型时，只需要确保能够容纳预期的最大数字即可，这样可以避免浪费不必要的存储空间。

对于浮点数值（如 3.1415927），可以选择表 2-4 中的数字类型。

表 2-4　MySQL 浮点类型

类　　型	数 值 范 围
float(*p*,*s*)	−3.402823466E+38～−1.175494351E-38 1.175494351E-38～3.402823466E+38
double(*p*,*s*)	−1.7976931348623157E+308～−2.2250738585072014E-308 2.2250738585072014E-308～1.7976931348623157E+308

当使用浮点类型时，可以指定其精度（小数点左边到右边所允许的数字总位数）和有效位（小数点右边所允许的数字位数），当然这不是必需的。这两个值在表 2-4 中由参数 *p* 和 *s* 指定。需要注意的是，如果数字位超过了该列所定义的精度或有效位，那么该列中存储的数据将会被四舍五入。例如，一个定义为 float(4,2)的列将会存储 4 位数字，其中两位在小数点左边，两位在小数点右边。因此，如果向该列添加数据 27.44 和 8.19 是允许的，但 17.8675 将会被四舍五入为 17.87，178.375 则会产生一个错误。

和整数类型一样，浮点列也可以被定义为 unsigned，但这里只是禁止列中存放负数，并没有改变该列所存储数据的范围。

2.3.3 时间数据

除了字符串和数字，处理信息还会经常用到日期或时间。这种类型的数据被称为时间型数据，下面为数据库中使用时间型数据的一些例子：

- 预计未来发生某个特定事件（比如发送客户订单）的日期；
- 客户订单实际上被发送的日期；
- 用户修改表中某行的日期与时间；
- 雇员的生日；
- 在数据仓库的 yearly_sales 表中，每行数据所关联的年份；
- 在自动装配线上完成一道流水线所消耗的时间。

MySQL 为所有这些情况都提供了合适的数据类型。表 2-5 显示了 MySQL 支持的时间数据类型。

表 2-5　MySQL 的时间类型

类　型	默 认 格 式	允 许 的 值
date	YYYY-MM-DD	1000-01-01～9999-12-31
datetime	YYYY-MM-DD HH:MI:SS	1000-01-01 00:00:00～9999-12-31 23:59:59
timestamp	YYYY-MM-DD HH:MI:SS	1970-01-01 00:00:00～2037-12-31 23:59:59
year	YYYY	1901～2155
time	HHH:MI:SS	−838:59:59～838:59:59

数据库服务器可以用各种方式存储这些时间数据，格式字符串（表 2-5 的第 2 列）用于指定显示这些数据在被提取时的显示方式，以及在插入或更新时间列时所需提供的日期字符串的格式。因此，如果需要向默认格式为 YYYY-MM-DD 的日期列中插入日期 2005 年 3 月 23 日，就必须使用字符串‘2005-03-23’。第 7 章完整地叙述了时间型数据是如何被构建和显示的。

提示

每种数据库服务器针对时间列所允许的日期范围各不相同。Oracle 数据库接受的日期从公元前 4712 年～公元 9999 年，SQL Server 则只能处理公元前 1753 年～公元 9999 年范围的日期（除非使用 SQL Server 2008 新增加的 datetime2 类型，它的日期范围是从公元前 1 年～公元 9999 年）。MySQL 的日期范围位于 Oracle 和 SQL Server 之间，它可以存储公元前 1000 年～公元 9999 年的日期。尽管对于大多数处理当前和将来事件的系统来说，这些差别并无意义，但是如果是存放历史日期就需要注意了。

表 2-6 描述了表 2-5 中日期格式的各个组成部分。

表 2-6　日期格式的组成部分

组 成 部 分	定　　义	范　　围
YYYY	年份，包括世纪	1000～9999
MM	月份	01 (January)～12 (December)
DD	日	01～31
HH	小时	00～23
HHH	小时（过去的)	−838～838
MI	分钟	00～59
SS	秒	00～59

下面介绍如何使用各种时间类型来实现前面提出的几个例子。

- 用于存储预计客户订单发送时间和雇员出生日期的列可以使用 date 类型，因为将调度发送订单的时间是不可能精确到分秒的，另外对某人出生的具体时间也不需要了解。
- 用于存放客户订单实际被发送时间的信息列可以为 datetime 类型，因为不仅需要跟踪发送的日期，记录发送的具体时间也是很重要的。
- 记录用户何时修改表中特定行可以使用 timestamp 类型。timestamp 保存的信息与 datetime 类型一样（包括年、月、日、分钟、秒），但 MySQL 服务器可以在向表中增加或修改数据行时自动为 timestamp 列产生当前的日期/时间。
- 只需要年份数据的年可以使用 year 类型。
- 存放完成某项任务所费时间的列可以使用 time 类型。这种类型的数据无法表示具体的日期，但本例只对完成任务所花费的小时/分钟/秒数感兴趣。此类信息还可以使用两个 datetime 列来存储（一个用于存放任务开始的日期/时间，另一个用于存放完成任务的日期/时间），通过两者之间的差值就可以计算所花费的时间，但显然

使用单个 time 列更为简单。

第 7 章将会阐述如何使用这些时间数据类型。

2.4 表的创建

现在你已经掌握了 MySQL 数据库中的各种数据类型，下面可以介绍如何在表定义中使用它们了。首先从定义一张存放个人信息的表开始。

2.4.1 第 1 步：设计

在开始设计一张表之前，最好对确定需要包含哪些信息进行一次头脑风暴。下面是经过我短暂思考后选择的用于描述个人的信息类型：

- 姓名；
- 性别；
- 出生日期；
- 地址；
- 最喜爱的食物。

显然该列表的信息并不完全，但在这里已经够用了。下一步是为它们指定列和数据类型，表 2-7 显示了初始时的设计。

表 2-7　person 表——初步结果

列	类　　型	允　许　值
name	varchar(40)	
gender	char(1)	M, F
birth_date	date	
address	varchar(100)	
favorite_foods	varchar(200)	

其中，name、address 和 favorite_foods 列的类型为 varchar，容许不同形式的数据条目，而 gender 列则只允许单个字母 M 或 F。birth_date 为日期类型，因为此处并不需要精确到具体的时间。

2.4.2 第 2 步：精化

在第 1 章已经介绍了规范化的概念，即在数据库设计时确保没有重复（外键除外）或复合列。再次观察 person 表，将会发现如下问题。

- name 列实际上是包含了姓氏和名字的复合对象。
- 可能存在多个人具有相同的名字、性别、生日等，而 person 表中并没有列来保证唯一性。
- address 列也是包含了街道、州/省名、县市名和邮政编码的复合对象。
- favorite_foods 列可以是包含 0、1 或更多条目的列表，因此最好为此数据创建一个独立的表，其中包含一个指向 person 表的外键，以便为每一种食物指明所归属的人员。

考虑到这些问题之后，表 2-8 列出了 person 表规范化后的结果。

表 2-8　person 表——第 2 步结果

列	类　　型	允　许　值
person_id	smallint (unsigned)	
first_name	varchar(20)	
last_name	varchar(20)	
gender	char(1)	M, F
birth_date	date	
street	varchar(30)	
city	varchar(20)	
state	varchar(20)	
country	varchar(20)	
postal_code	varchar(20)	

现在 person 表已经具有了主键(person_id)来保证唯一性,下一步便是建立 favorite_food 表，其中包含一个指向 person 表的外键。表 2-9 显示了结果。

表 2-9　favorite_food 表

列	类　　型
person_id	smallint (unsigned)
food	varchar(20)

person_id 和 food 列构成了 favorite_food 表的主键，并且 person_id 列也是 person 表的外键。

这些设计足够了吗?

将 favorite_foods 列移到 person 表之外肯定是个好主意,但这样就毫无问题了吗？例如，有两个人都喜欢意大利面，其中一人记录的是“pasta”，而另一个人则写下“spaghetti”

（pasta 和 spaghetti 都是指意大利面，译注），该怎么办呢？它们不都是同一种东西吗？为了防止此类问题发生，或许需要人们从列表项中选择他们喜爱的食物（而不是手动输入），这种情况下，应当创建包含 food_id 和 food_name 列的 food 表，然后修改 favorite_food 表，使其包含一个指向 food 表的外键。尽管这种设计是完全规范化的，但是如果你只是想简单地存储用户所输入的食物名称，那么可以保持原有的表设计。

2.4.3 第 3 步：构建 SQL 方案语句

现在已经完成了这两张表的设计，它们分别保存了人员和他们喜爱的食物信息，下一步就是生成在数据库中创建表的 SQL 语句。创建 person 表的语句如下。

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  gender CHAR(1),
  birth_date DATE,
  street VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

上述语句除了最后一项外，其他各列的含义都是显而易见的。在定义表时，需要向数据库指明哪个列或哪些列作为表的主键，通过为表建立一个约束（constraint）可以做到这一点。可以在表定义中增加多种类型的表约束，上述语句中的约束为主键约束，它被创建在 person_id 列上并被命名为 pk_person。

下面继续讨论有关约束的话题，对于 person 表来说，还有另一种类型的约束也是十分有用的。在表 2-7 中，第三列只接受特定的值（对于性别列来说，只能为‘M’和‘F’），这时可以为它增加一个检查约束，以限制该列只存放被允许的值。MySQL 允许在定义列时关联一个检查约束，如下所示：

```
gender CHAR(1) CHECK (gender IN ('M','F')),
```

尽管在大多数数据库服务器中检查约束能够如所期望的那样工作，但是对于 MySQL 来说，虽然允许定义检查约束，但并不强制使用它。实际上，MySQL 提供了另一种名为 enum 的字符数据类型，它可以将检查约束与数据类型定义融合到一起，下面提供此方式的 gender 列定义：

```
gender ENUM('M','F'),
```

下面是重新定义了的 person 表，其中使用 enum 作为 gender 列的数据类型：

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  gender ENUM('M','F'),
  birth_date DATE,
  street VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

本章后面将会介绍向列中添加违反检查约束（在 MySQL 中为枚举值）的数据会造成的结果。

现在可以在 mysql 的命令行工具里运行 create table 语句了，如下所示。

```
mysql> CREATE TABLE person
    -> (person_id SMALLINT UNSIGNED,
    ->  fname VARCHAR(20),
    ->  lname VARCHAR(20),
    ->  gender ENUM('M','F'),
    ->  birth_date DATE,
    ->  street VARCHAR(30),
    ->  city VARCHAR(20),
    ->  state VARCHAR(20),
    ->  country VARCHAR(20),
    ->  postal_code VARCHAR(20),
    ->  CONSTRAINT pk_person PRIMARY KEY (person_id)
    -> );
 Query OK, 0 rows affected (0.27 sec)
```

在处理完 create table 语句之后，MySQL 服务器将会返回消息“Query OK,0 rows affected,”，表明该语句没有语法错误。如果想确认 person 表实际上是否被创建，那么可以使用 describe 命令（或其简写 desc）来检查表定义。

```
mysql> DESC person;
+-------------+----------------------+------+-----+---------+-------+
| Field       | Type                 | Null | Key | Default | Extra |
+-------------+----------------------+------+-----+---------+-------+
| person_id   | smallint(5) unsigned |      | PRI | 0       |       |
| fname       | varchar(20)          | YES  |     | NULL    |       |
| lname       | varchar(20)          | YES  |     | NULL    |       |
| gender      | enum('M','F')        | YES  |     | NULL    |       |
| birth_date  | date                 | YES  |     | NULL    |       |
| street      | varchar(30)          | YES  |     | NULL    |       |
| city        | varchar(20)          | YES  |     | NULL    |       |
```

```
| state       | varchar(20)          | YES  |     | NULL    |       |
| country     | varchar(20)          | YES  |     | NULL    |       |
| postal_code | varchar(20)          | YES  |     | NULL    |       |
+-------------+----------------------+------+-----+---------+-------+
10 rows in set (0.06 sec)
```

describe 输出的第 1 列和第 2 列的含义是显而易见的，第 3 列显示该列是否允许在插入数据时被省略。现在对这一点进行简要的讨论（参见下面的“什么是 Null”小标题），在第 4 章中会有更完整地介绍。第 4 列显示该列是否作为键值（主键或外键），本例中 person_id 列被标记为主键。第 5 列显示如果在插入数据时忽略该列，是否向其提供默认值。本例中 person_id 列的默认值为 0，不过它只会起一次作用，因为 person 表中每一行数据在该列都必须包含一个唯一的值（因为它是主键）。第 6 列（“Extra”）显示该列附加的说明信息。

什么是 Null

某些情况下，在向表中插入数据时，无法为其中某一列提供具体的值。例如，当增加一条客户订单数据时，ship_date 列还不能确定，此时该列被设置为 null（注意我并没有说它等于 null），以指明该值的缺失。null 被用于各种不能赋值的情况，例如:

- 业务上不可行;
- 不知道应赋何值;
- 集合为空。

在设计表时，可以指定哪些列允许为 null（默认做法），哪些列不能为 null（通过在类型定义后面加上 not null 关键字指定）。

现在已经创建完 person 表，接下来是创建 favorite_food 表。

```
mysql> CREATE TABLE favorite_food
    -> (person_id SMALLINT UNSIGNED,
    ->  food VARCHAR(20),
    ->  CONSTRAINT pk_favorite_food PRIMARY KEY (person_id, food),
    ->  CONSTRAINT fk_fav_food_person_id FOREIGN KEY (person_id)
    ->    REFERENCES person (person_id)
    -> );
Query OK, 0 rows affected (0.10 sec)
```

除了有以下不同之外，它与前面 person 表的 create table 语句十分类似。

- 由于一个人可能有多种喜爱的食物（当然这也是创建本表的原因），仅靠 person_id 列不能保证表数据的唯一性，因此本表的主键包含两列：person_id 和 food。
- favorite_food 包含了另一种类型的约束，即外键约束，它限制了 favorite 表中 person_id 列的值只能够来自 person 表。通过这种约束，使得当 person 表中没有 person_id 为 27 的记录时，向 favorite_food 表中增加 person_id 为 27、喜爱食物为

比萨的数据行是不可能的。

提示

如果在首次建表时忘了创建外键约束，那么可以在后面通过 alter table 语句添加。

在执行完 create table 语句后，使用 describe 命令可以显示下面的结果：

```
mysql> DESC favorite_food;
+-------------+----------------------+------+-----+---------+------+
| Field       | Type                 | Null | Key | Default | Extra|
+-------------+----------------------+------+-----+---------+------+
| person_id   | smallint(5) unsigned|      | PRI |    0    |      |
| food        | varchar(20)          |      | PRI |         |      |
+-------------+----------------------+------+-----+---------+------+
```

现在数据库里已经包含该表了，接下来向其中添加一些数据。

2.5 操作与修改表

现在已经有了 person 表和 favorite_food 表，可以开始研究 4 种 SQL 数据语句（insert、update、delete 和 select）了。

2.5.1 插入数据

现在 person 表和 favorite_food 表中还没有任何数据，因此在这 4 种 SQL 数据语句中，我们首先讨论 insert 语句。下面是 insert 语句的 3 个主要组成部分：

- 所要插入数据的表的名称；
- 表中需要使用的列的名称；
- 需要插入到列的值。

实际上，向表中增加的数据并不需要覆盖所有列（除非表中所有列都被定义为 not null）。某些情况下，在最初的 insert 语句中并不包含部分列的值，而是之后通过 update 语句对其进行更新。在另一些情况中，对于特定的数据行来说，某列的值可能始终为 null（比如当客户订单在发送之前被取消，那么 ship_date 列就不再需要赋值了）。

生成数字型主键数据

在向 person 表中插入数据之前，先讨论一下数字型主键的生成机制是很有帮助的。除了随机选择数字外，还可以有以下两个常用的选择：

- 查看表中当前主键的最大值，并加 1；
- 让数据库服务器自动提供。

尽管第一种选择看起来很有效，但在多用户环境下可能会发生错误，因为两个用户或许会在同一时间访问表，并产生两个相同的主键值。实际上，市面上所有的数据库服务器都提供了一个安全、健壮的产生数字型主键的方法。在一些服务器中，如 Oracle 数据库，使用独立的方案对象，即序列号（sequence）；而在 MySQL 中，只需简单地为主键列打开自增（auto-increment）特性。一般情况下，应该在表创建时就进行此项工作，下面就此机会再介绍另一种 SQL 方案语句——alter table，它用于修改已存在的表定义：

```
ALTER TABLE person MODIFY person_id SMALLINT UNSIGNED AUTO_INCREMENT;
```

该语句实质上重新定义了 person 表的 person_id 列，现在如果再次使用 describe 命令，那么会看到 person_id 的“Extra”列中列出了自增特性。

```
mysql> DESC person;
+------------+--------------------+-----+----+--------+--------------+
| Field      | Type               | Null| Key |Default |Extra         |
+------------+--------------------+-----+----+--------+--------------+
| person_id  |smallint(5)unsigned |     | PRI |NULL    |auto_increment|
| .          |                    |     |    |        |              |
| .          |                    |     |    |        |              |
| .          |                    |     |    |        |              |
```

当向 person 表插入数据时，可以简单地将 person_id 列赋值为 null，MySQL 将会向该列提供下一个可用的主键数字（默认情况下，MySQL 从 1 开始自增）。

insert 语句

现在一切就绪了，该是向表中添加一些数据的时候了。下面的语句在 person 表中为 William Turner 创建了一行：

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date)
    -> VALUES (null, 'William','Turner', 'M', '1972-05-27');
Query OK, 1 row affected (0.01 sec)
```

运行结果（“Query OK, 1 row affected”）表明语句的语法正确，并且有 1 行数据已被添加到数据库（因为它是一条 insert 语句）。通过 select 语句可以查看这条数据：

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.06 sec)
```

如上所示，MySQL 服务器为主键产生的值为 1。因为现在 person 表中只有 1 条数据，

所以此处省略了查询条件，而只是简单地获取表中所有的数据。如果表中的数据多于 1 行，那么可以使用 where 子句指定想要提取的数据，即 person_id 列为 1 的行：

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE person_id = 1;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|       1   | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

该查询指定的是特定的主键值，实际上还可以使用表中任意列来搜索数据行，比如下面的查询用于查找 lname 列为‘Turner’的行：

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE lname = 'Turner';
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

在继续讨论之前，需要注意前面 insert 语句中的几个地方。

- 并没有为任何地址列赋值。这没有关系，因为这些列允许为 null。
- 为 birth_date 列提供的值为字符串，只要符合表 2-5 列出的格式， MySQL 就会自动将字符串转换为日期类型。
- 语句中提交的列数据必须在列数和类型上都符合表定义。如果表中有 7 列，但只提供了 6 个值，或者所提供的值不能被转换为对应列所要求的数据类型，那么将会接受到一个错误。

William 还提供了关于他最喜爱的 3 种食物的信息，因此还需要 3 条插入语句以保存他的食物偏好。

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'pizza');
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'cookies');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'nachos');
Query OK, 1 row affected (0.01 sec)
```

下面获取 William 的喜爱食物列表，并使用 order by 子句按字母顺序排列：

```
mysql> SELECT food
    -> FROM favorite_food
    -> WHERE person_id = 1
    -> ORDER BY food;
+---------+
| food    |
+---------+
| cookies |
| nachos  |
| pizza   |
+---------+
3 rows in set (0.02 sec)
```

order by 子句指示服务器如何对查询返回的数据进行排序。如果不使用 order by 子句，对表中数据获取的次序并不做任何保证。

为了让 William 不感到孤单，可以再次执行 insert 语句向 person 表中增加 Susan Smith：

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date,
    ->   street, city, state, country, postal_code)
    -> VALUES (null, 'Susan','Smith', 'F', '1975-11-02',
    ->   '23 Maple St.', 'Arlington', 'VA', 'USA', '20220');
 Query OK, 1 row affected (0.01 sec)
```

由于 Susan 大方地提供了她的地址，上面的 insert 语句包含的列数比插入 William 数据时多 5 列。如果再次查询该表，将会看到 Susan 所在行的主键已经被自动赋值为 2：

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|       1   | William | Turner | 1972-05-27 |
|       2   | Susan   | Smith  | 1975-11-02 |
+-----------+---------+--------+------------+
2 rows in set (0.00 sec)
```

可以获取 XML 格式的数据吗？

如果你正在使用 XML 数据，就会很高兴地看到大多数数据库都已经提供了简便的方法从查询结果中生成 XML。例如，对于 MySQL，可以在调用 mysql 工具时使用--xml 选项，这样查询的输出将会自动使用 XML 格式化。下面演示如何获取 XML 文档格式的 favorite_food 数据：

```
C:\database> mysql -u lrngsql -p --xml bank
Enter password: xxxxxx
Welcome to the MySQL Monitor...
```

```
Mysql> SELECT * FROM favorite_food;
<?xml version="1.0"?>

<resultset statement="select * from favorite_food"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
        <field name="person_id">1</field>
        <field name="food">cookies</field>
  </row>
  <row>
        <field name="person_id">1</field>
        <field name="food">nachos</field>
  </row>
  <row>
        <field name="person_id">1</field>
        <field name="food">pizza</field>
  </row>
</resultset>
3 rows in set (0.00 sec)
```

在 SQL Server 数据库中，无需配置命令行工具，只是在每个查询末尾增加 for xml 子句即可:

```
SELECT * FROM favorite_food
FOR XML AUTO, ELEMENTS
```

2.5.2 更新数据

在 William Turner 的数据被添加到表中时，insert 语句中忽略了他的地址列。下面演示如何通过 update 语句更新这些列上的数据：

```
mysql> UPDATE person
   -> SET street = '1225 Tremont St.',
   ->   city = 'Boston',
   ->   state = 'MA',
   ->   country = 'USA',
   ->   postal_code = '02138'
   -> WHERE person_id = 1;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

服务器响应了两行信息："Rows matched: 1"条目提示 where 子句中的条件匹配了表中的 1 行数据，而"Changed: 1"条目提示表中有 1 行数据被修改。其中，where 子句指定了 William 所在行的主键值，因此所修改的数据肯定与预期一样。

根据 where 子句中的条件，在单个语句中修改多个数据行是可能的。例如，假设 where 语句如下：

```
WHERE person_id < 10
```

由于 William 和 Susan 的 person_id 值都小于 10，因此他们所在行都将被修改。如果省略整个 where 子句，那么 update 语句将修改表中的每一行。

2.5.3 删除数据

现在看来 William 和 Susan 在一起相处的并不好，因此他们中的一个必须离开。既然 William 是先加入的，那么可以将 Susan 列为 delete 语句的对象：

```
mysql> DELETE FROM person
    -> WHERE person_id = 2;
Query OK, 1 row affected (0.01 sec)
```

同样地，主键再次被用于识别所感兴趣的行，因此表中只会有 1 行数据被删除。与 update 语句类似，根据 where 子句中的条件，同时删除多行数据是允许的，省略 where 子句则会删除表中的所有行。

2.6 导致错误的语句

到目前为止，本章所有的 SQL 数据语句都符合标准格式并运行良好。但对于 person 和 favorite_food 的表定义来说，在删除或修改数据时可能会出现很多运行错误，本节主要讨论一些常见错误，以及 MySQL 服务器是如何应答它们的。

2.6.1 主键不唯一

由于表定义中创建了主键约束，因此 MySQL 将会确保重复主键不会被插入到数据表中。下列语句忽略了 person_id 列的自增性质，直接将 person 表中 person_id 值设为 1：

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date)
    -> VALUES (1, 'Charles','Fulton', 'M', '1968-01-15');
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

只要保证 person_id 列的值不同，就可以创建两个具有完全相同的姓名、地址、出生日期等条目的数据行，至少在目前的方案对象中是不受限制的。

2.6.2 不存在的外键

favoriate_food 的表定义在 person_id 列上创建了外键，该约束确保 favorite_food 表中所输入的 person_id 列的值都存在于 person 表中。下面显示了创建行时违背这一约束的结果：

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (999, 'lasagna');
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint
```

```
fails ('bank'.'favorite_food', CONSTRAINT 'fk_fav_food_person_id'
FOREIGN KEY
('person_id') REFERENCES 'person' ('person_id'))
```

在此情况下，由于 favorite_food 表的部分数据依赖于 person 表，因此可以将 favoriate_food 表视为子表，而将 person 表视为父表，如果需要同时向两个表中输入数据，那么在向 favoriate_food 表输入之前，必须先在父表中创建一行。

提示

外键约束只能在使用 InnoDB 存储引擎创建表时才起作用。第 12 章将会讨论 MySQL 的存储引擎。

2.6.3 列值不合法

Person 表中的 gender 列被限制为只容许‘M’（男性）或‘F’（女性）值，如果试图将该列设为任何其他值，都将会收到如下响应：

```
mysql> UPDATE person
    -> SET gender = 'Z'
    -> WHERE person_id = 1;
ERROR 1265 (01000): Data truncated for column 'gender' at row 1
```

该错误信息有点令人迷惑，但仍能显示出服务器并不乐意接受所提供的 gender 列值。

2.6.4 无效的日期转换

如果构建一个用于产生日期列的字符串，那么该字符串必须符合要求的格式，否则将会接受到一个错误。下面的例子中，所使用的日期格式没有与默认的日期格式“YYYY-MM-DD”相匹配：

```
mysql> UPDATE person
    -> SET birth_date = 'DEC-21-1980'
    -> WHERE person_id = 1;
ERROR 1292 (22007): Incorrect date value: 'DEC-21-1980' for column
'birth_date'
at row 1
```

通常情况下，显式地指定字符串格式是比依赖默认格式更好的做法。下面的语句使用 str_to_date 函数指定所用字符串格式：

```
mysql> UPDATE person
    -> SET birth_date = str_to_date('DEC-21-1980' , '%b-%d-%Y')
    -> WHERE person_id = 1;
Query OK, 1 row affected (0.12 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

这样做不仅使服务器感到满意，William 也会同样开心（我们在不需要进行昂贵的美容手术的情况下，使他年轻了八岁！）。

提示

在本章前面介绍各种时间数据类型时已经展示了日期格式字符串，比如"YYYY-MM-DD"。大多数数据库服务器采用这种风格的格式，但 MySQL 使用%Y 来指定 4 位数字的年份。下面为 MySQL 中将字符串转换为 datetime 型值时可能用到的格式:

```
%a 星期几的简写，比如 Sun、Mon、 ...
%b 月名称的简写，比如 Jan、Feb、...
%c 月份的数字形式(0..12)
%d 日在月中的次序(00..31)
%f 毫秒数(000000..999999)
%H 24 时格式中的小时(00..23)
%h 12 时格式中的小时(01..12)
%i 小时中的分钟(00..59)
%j 一年中天的次序(001..366)
%M 完成的月名称(January..December)
%m 月份的数字表示
%p AM 或 PM
%s 秒数(00..59)
%W 完整的星期名(Sunday..Saturday)
%w 天在星期中的次序 (0=周日..6=周六)
%Y 4 位数字的年份
```

2.7 Bank 方案

在本书的剩余部分，需要为某公共银行建立一组数据表模型，其中包括 employee、branch、account、custmoer、product 和 transaction 等表。如果你已经按照本章开始时介绍的步骤载入 MySQL 服务器并创建示例数据，就已经创建好完整的方案和例子数据了。在附录 A 中包含了各个表的字段以及相互关系的图示。

表 2-10 显示了 bank 方案中各数据表的简短定义。

表 2-10 Bank 方案定义

表 名	定 义
account	为特定顾客开放的特定产品
branch	开展银行交易业务的场所
business	公司顾客(customer 表的子类型)
customer	与银行有业务来往的个人或公司
department	执行特定银行职能的雇员分组
employee	银行的工作人员
individual	个人顾客 (customer 表的子类型)

续表

表　　名	定　　义
officer	允许为公司顾客发起商务交易的人
product	向顾客提供的银行服务
product_type	具备相似功能的产品的分组
transaction	改变账户余额的操作

在进行相关 SQL 实验时，可以根据自己的需要自由地使用这些表，还可以增加新的表以扩展银行的业务功能。如果在一段时间后示例数据被修改得面目全非，可以先丢弃数据库，再使用下载文件重新创建示例数据库。

如果需要查看数据库中可用的表，可以使用 show tables 命令，如下所示。

```
mysql> SHOW TABLES;
+----------------+
| Tables_in_bank |
+----------------+
| account        |
| branch         |
| business       |
| customer       |
| department     |
| employee       |
| favorite_food  |
| individual     |
| officer        |
| person         |
| product        |
| product_type   |
| transaction    |
+----------------+
13 rows in set (0.10 sec)
```

除了银行方案中的 11 个表，该列表还包含了本章创建的两个表：person 和 favoriate_达式 food。这些表在后面的章节中将不再使用，因此可以使用下面的命令删除它们：

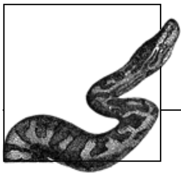
```
mysql> DROP TABLE favorite_food;
Query OK, 0 rows affected (0.56 sec)
mysql> DROP TABLE person;
Query OK, 0 rows affected (0.05 sec)
```

如果需要查看表中的每个列，可以使用 describe 命令。下面的例子给出了 customer 表的 describe 输出：

```
mysql> DESC customer;
+--------------+-------------+------+-----+---------+------------+
| Field        | Type        | Null | Key | Default | Extra      |
```

```
+-------------+-------------+------+-----+---------+------------+
| cust_id      |int(10)unsigned| NO   | PRI |NULL     |auto_increment|
| fed_id       |varchar(12)    | NO   |     | NULL    |            |
| cust_type_cd |enum('I','B')  | NO   |     | NULL    |            |
| address      |varchar(30)    | YES  |     | NULL    |            |
| city         |varchar(20)    | YES  |     | NULL    |            |
| state        |varchar(20)    | YES  |     | NULL    |            |
| postal_code  |varchar(10)    | YES  |     | NULL    |            |
+-------------+-------------+------+-----+---------+------------+
7 rows in set (0.03 sec)
```

对示例数据库越熟悉，对本书后面章节的例子与概念就会理解得更深刻。



第 3 章

查询入门

到此，本书在前面两章已经介绍了一些数据库查询的例子（那些 select 语句），本章将详细讨论 select 语句的各组成部分，以及它们之间是如何相互作用的。

3.1 查询机制

在解析 select 语句之前，或许探讨一下 MySQL 服务器执行查询的机制也是一件有趣的事（当然对其他数据库服务器也是如此）。首先打开 mysql 命令行工具，然后使用用户名和密码登录（如果 MySQL 服务器运行在另一台计算机上，还需要提供主机名）。一旦服务器通过了对用户名和密码的验证，则为用户创建一个数据库连接。该连接从应用程序（此处即 mysql 工具）发出请求后一直保持，直到应用程序释放连接（比如输入 quit 命令）或者服务器关闭连接（比如服务器关机）。MySQL 服务器会给每个连接赋予一个标识符，它会在首次登录时显示：

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 6.0.3-alpha-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

本例中，连接 ID 是 11。在发生异常情况时，该信息可能会给数据库管理员带来帮助，比如手动结束运行几个小时的有问题的查询。

在服务器验证完用户名和密码，并且创建连接后，用户就可以执行查询了（当然也包括其他 SQL 语句）。每当查询被发送到服务端时，服务器在执行语句之前将会进行下面的检查：

- 用户是否有权限执行该语句？
- 用户是否有权限访问目标数据？

- 语句的语法是否正确?

如果查询语句通过了这 3 个测试，就会被传递给查询优化器，它负责为查询找到最有效率的执行方式。优化器通常会做诸如确定 from 子句后面各表的连接顺序，或是可以使用哪些索引之类的工作，然后选择一个执行方案，以供服务器执行该查询。

> **提示**
> 对于很多读者来说，理解和影响数据库服务器选择查询执行方案是一个极具吸引力的主题。对于使用 MySQL 的读者，建议阅读 Baron Schwartz 等人著作的 High Performance MySQL 一书，其中介绍了如何建立索引、分析执行方案、通过查询提示（query hints）分析优化器以及调优服务器的初始参数等。如果你使用的是 Oralce 或 SQL Server 数据库，那么可供学习的性能调优书籍就更多了。

当服务器执行完查询以后，将会向调用程序（再次提示：本例中即 mysql 工具）返回一个结果集。在第 1 章中已经提到，结果集实际上也是一种包含行与列的表。如果查询并没有找到任何结果，那么 mysql 工具将会在其后显示一条提示消息，例如：

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
-> WHERE lname = 'Bkadfl';
Empty set(0.00 sec)
```

如果查询返回了 1 行或多行记录，那么 mysql 工具将会使用列名和-、|和+等符号组成的边框将结果格式化输出，例如：

```
mysql> SELECT fname, lname
    -> FROM employee;
+----------+-----------+
| fname    | lname     |
+----------+-----------+
| Michael  | Smith     |
| Susan    | Barker    |
| Robert   | Tyler     |
| Susan    | Hawthorne |
| John     | Gooding   |
| Helen    | Fleming   |
| Chris    | Tucker    |
| Sarah    | Parker    |
| Jane     | Grossman  |
| Paula    | Roberts   |
| Thomas   | Ziegler   |
| Samantha | Jameson   |
| John     | Blake     |
| Cindy    | Mason     |
| Frank    | Portman   |
| Theresa  | Markham   |
```

```
| Beth      | Fowler     |
| Rick      | Tulman     |
+----------+-----------+
18 rows in set (0.00 sec)
```

该查询返回了 employee 表中所有雇员的姓氏和名字，在显示最后一行数据之后，mysql 工具会显示一条消息，以提示一共返回了多少行。

3.2 查询语句

select 语句由几个组件或者说子句构成。不过在 MySQL 中，只有一种子句是必不可少的（select 子句），通常的查询语句会至少包含 6 个子句中的 2～3 个。表 3-1 列出了用于不同目的的各个子句。

表 3-1 query 子句

子句名称	使用目的
select	确定结果集中应该包含哪些列
from	指明所要提取数据的表，以及这些表是如何连接的
where	过滤不需要的数据
group by	用于对具有相同列值的行进行分组
having	过滤掉不需要的组
order by	按一个或多个列，对最后结果集中的行进行排序

表 3-1 中显示的所有子句都属于 ANSI 标准，此外，MySQL 还有一些特有的字句，将在附录 B 中进行介绍。本章下面各节将侧重讨论这 6 种主要查询子句的使用方法。

3.3 select 子句

尽管 select 子句是 select 语句中的第一个组成部分，但实际上在数据库服务中，它是最后被评估的。因为在确定结果集最后包含哪些列之前，必须先要知道结果集所有可能包含的列。因此为了更好地理解 select 子句的角色，首先需要了解一下 from 子句。下面开始一个查询：

```
mysql> SELECT *
    -> FROM department;
+---------+----------------+
| dept_id | name           |
+---------+----------------+
|       1 | Operations     |
|       2 | Loans          |
|       3 | Administration |
```

```
+---------+----------------+
3 rows in set (0.04 sec)
```

在此查询中，from 子句只列出了一个表（department），并且 select 子句指示在结果集中需要包含所有 departement 表中的列（通过*号表示）。该查询的含义可以如下表达：

显示 department 表中所有的行和列。

除了通过星号来指代所有列之外，还可以显示使用中感兴趣的列名，例如：

```
mysql> SELECT dept_id, name
    -> FROM department;
+---------+----------------+
| dept_id | name           |
+---------+----------------+
|      1  | Operations     |
|      2  | Loans          |
|      3  | Administration |
+---------+----------------+
3 rows in set (0.01 sec)
```

结果与前一个查询完全一样，因为 department 表中所有的列（dept_id 和 name）都在 select 子句中被显式列了出来。当然也可以选择只获取 department 表中各列的一个子集，例如：

```
mysql> SELECT name
    -> FROM department;
+----------------+
| name           |
+----------------+
| Operations     |
| Loans          |
| Administration |
+----------------+
3 rows in set (0.00 sec)
```

因此 select 子句的作用可以概括如下：

select 子句用于在所有可能的列中，选择查询结果集要包含哪些列。

如果数据库限制了只能返回 from 子句后面各表中所包含的列，就显得相当乏味了。幸运的是，我们可以在 select 子句中加上一些“调料”，例如：

- 字符，比如数字或字符串；
- 表达式，比如 transaction.amount*-1；
- 调用内建函数，比如 ROUND(transaction.amount,2)；
- 用户自定义的函数调用。

下面展示了对于 employee 表，如何在查询语句中使用列名、字符、表达式和内建函数调用：

```
mysql> SELECT emp_id,
    -> 'ACTIVE',
    -> emp_id * 3.14159,
    -> UPPER(lname)
    -> FROM employee;
+-------+--------+---------------+------------+
| emp_id| ACTIVE |emp_id * 3.14159| UPPER(lname)|
+-------+--------+---------------+------------+
|     1 | ACTIVE |       3.14159 | SMITH      |
|     2 | ACTIVE |       6.28318 | BARKER     |
|     3 | ACTIVE |       9.42477 | TYLER      |
|     4 | ACTIVE |      12.56636 | HAWTHORNE  |
|     5 | ACTIVE |      15.70795 | GOODING    |
|     6 | ACTIVE |      18.84954 | FLEMING    |
|     7 | ACTIVE |      21.99113 | TUCKER     |
|     8 | ACTIVE |      25.13272 | PARKER     |
|     9 | ACTIVE |      28.27431 | GROSSMAN   |
|    10 | ACTIVE |      31.41590 | ROBERTS    |
|    11 | ACTIVE |      34.55749 | ZIEGLER    |
|    12 | ACTIVE |      37.69908 | JAMESON    |
|    13 | ACTIVE |      40.84067 | BLAKE      |
|    14 | ACTIVE |      43.98226 | MASON      |
|    15 | ACTIVE |      47.12385 | PORTMAN    |
|    16 | ACTIVE |      50.26544 | MARKHAM    |
|    17 | ACTIVE |      53.40703 | FOWLER     |
|    18 | ACTIVE |      56.54862 | TULMAN     |
+-------+--------+---------------+------------+
18 rows in set (0.05 sec)
```

在本书后面将会详细介绍上面使用的表达式和内建函数，本例已经大致演示了在 select 子句中可以包含哪些元素。如果只是需要执行一个内建函数或对简单的表达式求值可以完全省略 from 子句，例如：

```
mysql> SELECT VERSION(),
    ->  USER(),
    ->  DATABASE();
+-----------------------+------------------+------------+
| version()             | user()           | database() |
+-----------------------+------------------+------------+
| 6.0.3-alpha-community | lrngsql@localhost | bank       |
+-----------------------+------------------+------------+
1 row in set (0.05 sec)
```

该查询只是简单地调用了 3 个内建函数，并没有从任何表中获取数据，因此不需要使用 from 子句。

3.3.1 列的别名

尽管 mysql 命令行工具为查询所返回的每个列提供了默认标签，但或许你希望使用自己定义的标签。如果表中的列名定义十分简短并且含义模糊，可以为该列赋予新标签。同样，如果在结果集中包含了根据表达式或内建函数调用产生的列，那么也可以为这些列定义一个标签。通过在 select 子句中的每个元素后面增加列别名可以实现此目的，下面对 employee 表的查询与前面的查询相似，只不过为其中的 3 列定义了列别名：

```
mysql> SELECT emp_id,
    ->  'ACTIVE' status,
    ->  emp_id * 3.14159 empid_x_pi,
    ->  UPPER(lname) last_name_upper
    -> FROM employee;
+-------+--------+-----------+-------------+
|emp_id | status | empid_x_pi |last_name_upper|
+-------+--------+-----------+-------------+
|     1 | ACTIVE |   3.14159  | SMITH        |
|     2 | ACTIVE |   6.28318  | BARKER       |
|     3 | ACTIVE |   9.42477  | TYLER        |
|     4 | ACTIVE |  12.56636  | HAWTHORNE    |
|     5 | ACTIVE |  15.70795  | GOODING      |
|     6 | ACTIVE |  18.84954  | FLEMING      |
|     7 | ACTIVE |  21.99113  | TUCKER       |
|     8 | ACTIVE |  25.13272  | PARKER       |
|     9 | ACTIVE |  28.27431  | GROSSMAN     |
|    10 | ACTIVE |  31.41590  | ROBERTS      |
|    11 | ACTIVE |  34.55749  | ZIEGLER      |
|    12 | ACTIVE |  37.69908  | JAMESON      |
|    13 | ACTIVE |  40.84067  | BLAKE        |
|    14 | ACTIVE |  43.98226  | MASON        |
|    15 | ACTIVE |  47.12385  | PORTMAN      |
|    16 | ACTIVE |  50.26544  | MARKHAM      |
|    17 | ACTIVE |  53.40703  | FOWLER       |
|    18 | ACTIVE |  56.54862  | TULMAN       |
+-------+--------+-----------+-------------+
18 rows in set (0.00 sec)
```

注意，表头中第二、三、四列显示的都是有意义的名称，而不是简单地显示产生该列的函数或表达式的名字，这是因为 select 子句中为它们指定了别名——status、empid_x_pi 和 last_name_upper。使用这些列名使输出的结果更易于理解，同时如果不是使用 mysql 工具，而是在 Java 或 C#中进行查询，这样做更易于编程实现。为了在子句中更清晰地表示别名，可以在这些别名前面加上关键字 as，例如：

```
mysql> SELECT emp_id,
    ->  'ACTIVE' AS status,
```

```
    ->   emp_id * 3.14159 AS empid_x_pi,
    ->   UPPER(lname) AS last_name_upper
-> FROM employee;
```

许多人认为加上可选的 as 关键字可以提高查询语句的可读性，不过本书选择在例子中不使用该关键字。

3.3.2 去除重复的行

在某些情况下，查询可能会返回重复的行数据，比如在提取所有 account 的 customer ID 时，将会出现下面的结果：

```
mysql> SELECT cust_id
    -> FROM account;
+---------+
| cust_id |
+---------+
|       1 |
|       1 |
|       1 |
|       2 |
|       2 |
|       3 |
|       3 |
|       4 |
|       4 |
|       4 |
|       5 |
|       6 |
|       6 |
|       7 |
|       8 |
|       8 |
|       9 |
|       9 |
|       9 |
|      10 |
|      10 |
|      11 |
|      12 |
|      13 |
+---------+
24 rows in set (0.00 sec)
```

因为某些客户可能具有多个账户，所以查询结果中多次包含了该客户的 ID。实际上你很可能只需要对每个客户显示一次 ID，而不是为 account 表中每一行都显示相应的客户 ID，这时可以在 select 关键字之后加上 distinct 关键字，例如：

```
mysql> SELECT DISTINCT cust_id
    -> FROM account;
```

```
+---------+
| cust_id |
+---------+
|      1  |
|      2  |
|      3  |
|      4  |
|      5  |
|      6  |
|      7  |
|      8  |
|      9  |
|     10  |
|     11  |
|     12  |
|     13  |
+---------+
13 rows in set (0.01 sec)
```

现在结果集中只包含了 13 行（每个独立的客户 ID 只出现一次），而不是 24 行（为每个 account 记录都显示一次）。

如果你并不需要服务器删除重复的数据，或者你确信结果集中不会包含重复记录，那么可以指定 ALL 关键字来替代 DISTINCT 关键字。不过事实上，ALL 关键字是系统默认的，不需要被显式地列出，因此大多数程序员不会在查询中加上 ALL 关键字。

警告

注意产生无重复的结果集需要首先对数据排序，这对于大的结果集来说是相当耗时的。因此不要为了确保去除重复行而随意地使用 DISTINCT，而是应该先了解所使用的数据是否可能包含重复行，以减少对 DISTINCT 的不必要的使用。

3.4 from 子句

上面的例子中已经介绍了包含单个表的 from 子句，尽管大多数 SQL 书籍都将 from 子句定义为 1 个或多个表的清单，但本书将会对其定义作以下拓展：

from 子句定义了查询中所使用的表，以及连接这些表的方式。

3.4.1 表的概念

当使用术语“表”的时候，大多数人联想到的是存储在数据库中关联行的集合，但这实际上只是表的类型之一而已。我们可以用更广泛的方式使用这个术语，去除其中蕴含的“被存储的数据”的含义，而仅仅考虑其关联行集合的含义。因此在对“表”的

宽泛的定义下，存在 3 种类型的表：

- 永久表（使用 create table 语句创建的表）；
- 临时表（子查询所返回的表）；
- 虚拟表（使用 create view 子句所创建的视图）。

这 3 种类型的表都可以在查询的 from 子句中使用。由于读者可能已经熟悉了在 from 子句中包含永久表，因此下面将主要讨论在如何在 from 子句中使用另外两种类型的表。

子查询产生的表

子查询指的是包含在另一个查询中的查询。子查询可以出现在 select 语句中的各个部分并且被包含在圆括号中。在 from 子句内，子查询的作用是根据其他查询子句（其中的 from 子句可以与其他表进行交互）产生临时表，下面是一个简单的例子：

```
mysql> SELECT e.emp_id, e.fname, e.lname
   -> FROM (SELECT emp_id, fname, lname, start_date, title
   ->      FROM employee) e;
+-------+----------+-----------+
|emp_id | fname    | lname     |
+-------+----------+-----------+
|     1 | Michael  | Smith     |
|     2 | Susan    | Barker    |
|     3 | Robert   | Tyler     |
|     4 | Susan    | Hawthorne |
|     5 | John     | Gooding   |
|     6 | Helen    | Fleming   |
|     7 | Chris    | Tucker    |
|     8 | Sarah    | Parker    |
|     9 | Jane     | Grossman  |
|    10 | Paula    | Roberts   |
|    11 | Thomas   | Ziegler   |
|    12 | Samantha | Jameson   |
|    13 | John     | Blake     |
|    14 | Cindy    | Mason     |
|    15 | Frank    | Portman   |
|    16 | Theresa  | Markham   |
|    17 | Beth     | Fowler    |
|    18 | Rick     | Tulman    |
+-------+----------+-----------+
18 rows in set (0.00 sec)
```

本例中，针对 employee 表的子查询返回 5 个列，而外围的查询获取其中 3 个列。在外围查询中，通过别名（本例中为 e）来引用子查询。这是一个关于子查询的简单而并不十分有用的例子，第 9 章将会对其进行详细讨论。

视图

视图是存储在数据字典中的查询，它的行为表现得像一个表，但实际上并不拥有任何数据（因此本书称之为虚拟表）。当发出一个对视图的查询时，该查询会被绑定到视图定义上，以产生最终被执行的查询。

下面首先定义一个查询 employee 表的视图，其中包含了一个对内建函数的调用：

```
mysql> CREATE VIEW employee_vw AS
    -> SELECT emp_id, fname, lname,
    ->   YEAR(start_date) start_year
    -> FROM employee;
Query OK, 0 rows affected (0.10 sec)
```

当视图被创建后，并没有产生或存储任何数据，服务器只是简单地保留该查询以供将来使用。现在既然该视图已经存在了，那么就可以对其发出查询请求，例如：

```
mysql> SELECT emp_id, start_year
    -> FROM employee_vw;
+--------+------------+
| emp_id | start_year |
+--------+------------+
|      1 |      2005  |
|      2 |      2006  |
|      3 |      2005  |
|      4 |      2006  |
|      5 |      2007  |
|      6 |      2008  |
|      7 |      2008  |
|      8 |      2006  |
|      9 |      2006  |
|     10 |      2006  |
|     11 |      2004  |
|     12 |      2007  |
|     13 |      2004  |
|     14 |      2006  |
|     15 |      2007  |
|     16 |      2005  |
|     17 |      2006  |
|     18 |      2006  |
+--------+------------+
18 rows in set (0.07 sec)
```

创建视图可能出于各种理由，比如对用户隐藏列、简化数据库设计等。

3.4.2 表连接

对 from 子句定义的第 2 种偏差（第 1 种指的是对表定义的理解，译者注）是：如果 from 子句中出现了多个表，那么要求同时包含各表之间的连接条件。这在 MySQL 数据库或

其他任何数据库中都不是必需的，但却是 ANSI 所建议的连接多个表的方法，也是在各种数据库服务器之间最具可移植性的做法。本书将会在第 5 章和第 10 章中对连接多个表进行深入介绍，而在此处只是提供一个简单的例子以满足读者的好奇心：

```
mysql> SELECT employee.emp_id, employee.fname,
    ->   employee.lname, department.name dept_name
    -> FROM employee INNER JOIN department
    ->   ON employee.dept_id = department.dept_id;
+--------+----------+-----------+----------------+
| emp_id | fname    | lname     | dept_name      |
+--------+----------+-----------+----------------+
|      1 | Michael  | Smith     | Administration |
|      2 | Susan    | Barker    | Administration |
|      3 | Robert   | Tyler     | Administration |
|      4 | Susan    | Hawthorne | Operations     |
|      5 | John     | Gooding   | Loans          |
|      6 | Helen    | Fleming   | Operations     |
|      7 | Chris    | Tucker    | Operations     |
|      8 | Sarah    | Parker    | Operations     |
|      9 | Jane     | Grossman  | Operations     |
|     10 | Paula    | Roberts   | Operations     |
|     11 | Thomas   | Ziegler   | Operations     |
|     12 | Samantha | Jameson   | Operations     |
|     13 | John     | Blake     | Operations     |
|     14 | Cindy    | Mason     | Operations     |
|     15 | Frank    | Portman   | Operations     |
|     16 | Theresa  | Markham   | Operations     |
|     17 | Beth     | Fowler    | Operations     |
|     18 | Rick     | Tulman    | Operations     |
+--------+----------+-----------+----------------+
18 rows in set (0.05 sec)
```

上面的查询需要同时显示 employee 表（emp_id、fname、lname）和 department 表（name）的数据，因此这两个表都包含在 from 子句中。连接两个表的机制（join 方式）是 employee 表中所存储的雇员部门信息。因此，数据库服务器使用 employee 表中的 dept_id 列值在 departement 表中查找关联的部门名称。两个表的连接条件由 from 子句中的 on 子句指定，本例中的连接条件为 ON employee.dept_id = department.dept_id。再次提示，第 5 章中对连接多个表进行了完整的讨论。

3.4.3 定义表别名

当在单个查询中连接多个表时，需要在 select、where、group by、having 以及 order by 子句中指明所引用的是哪个表。有两种在 from 子句之外引用表的方式：

- 使用完整的表名称，如 employee.emp_id；
- 为每个表指定别名，并在查询中需要的地方使用该别名。

上一个查询中，在 select 子句和 on 子句内使用了完整的表名，下面使用定义表别名的方式实现同样的查询：

```
SELECT e.emp_id, e.fname, e.lname,
  d.name dept_name
FROM employee e INNER JOIN department d
  ON e.dept_id = d.dept_id;
```

注意在 from 子句中，employee 表被命名为 e、department 表被命名为 d。这些别名在定义连接条件时被 on 子句使用，同时在 select 子句中为结果集指定所要包含的列时也用到了它们。我认为使用别名不会给紧凑的查询语句造成混乱（只要别名的选择是合理的）。此外，还可以在别名的前面使用 as 关键字，与前文提到的对列别名的使用方式类似：

```
SELECT e.emp_id, e.fname, e.lname,
  d.name dept_name
FROM employee AS e INNER JOIN department AS d
  ON e.dept_id = d.dept_id;
```

根据我的观察，大概一半的数据库开发者对列或表的别名使用 as 关键字，而另一半并不采用这种做法。

3.5 where 子句

到目前为止，本章中的查询示例都是从 employee、department 或 account 表中选择所有数据（除了关于使用 distinct 的一个例子）。但在大多数情况下，并不需要提取表中的每一行，而是希望使用某种方式过滤掉不感兴趣的行，这就是 where 子句的作用。

where 子句用于在结果集中过滤掉不需要的行。

举个例子，如果需要查询 employee 表，但只想获取头衔为 head teller 的雇员数据，那么就可以在查询的 where 子句中进行指定，这样结果中将只包含 4 个符合条件的行：

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller';
+--------+---------+---------+------------+-------------+
| emp_id | fname   | lname   | start_date | title       |
+--------+---------+---------+------------+-------------+
|      6 | Helen   | Fleming | 2008-03-17 | Head Teller |
|     10 | Paula   | Roberts | 2006-07-27 | Head Teller |
|     13 | John    | Blake   | 2004-05-11 | Head Teller |
|     16 | Theresa | Markham | 2005-03-15 | Head Teller |
+--------+---------+---------+------------+-------------+
4 rows in set (1.17 sec)
```

在本例中，where 子句过滤掉 18 个雇员行中的 14 行。该子句中只包含了一个过滤条件，

但在需要时可以同时包含更多的条件，它们之间使用操作符 and、or 或者 not 分隔（参见第 4 章，其中对 where 子句和过滤条件进行了完整的讨论）。下面是对前一个查询的扩展，它包含的第 2 个条件指明了只包括在 2006 年 1 月之后入职的雇员：

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller'
    ->   AND start_date > '2006-01-01';
+--------+-------+---------+------------+-------------+
| emp_id | fname | lname   | start_date | title       |
+--------+-------+---------+------------+-------------+
|     6  | Helen | Fleming | 2008-03-17 | Head Teller |
|    10  | Paula | Roberts | 2006-07-27 | Head Teller |
+--------+-------+---------+------------+-------------+
2 rows in set (0.01 sec)
```

第 1 个条件（title='Head Teller'）过滤掉 18 行中的 14 行，而第 2 个条件（start_date > ‘2006-01-01’）又过滤掉两行，因此在最后的结果集中只包含了剩下的两行。下面看看如果把两个条件中的 and 操作符改成 or 后会发生什么变化：

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller'
    ->   OR start_date > '2006-01-01';
+-------+----------+-----------+------------+--------------------+
|emp_id | fname    | lname     | start_date | title              |
+-------+----------+-----------+------------+--------------------+
|     2 | Susan    | Barker    | 2006-09-12 | Vice President     |
|     4 | Susan    | Hawthorne | 2006-04-24 | Operations Manager |
|     5 | John     | Gooding   | 2007-11-14 | Loan Manager       |
|     6 | Helen    | Fleming   | 2008-03-17 | Head Teller        |
|     7 | Chris    | Tucker    | 2008-09-15 | Teller             |
|     8 | Sarah    | Parker    | 2006-12-02 | Teller             |
|     9 | Jane     | Grossman  | 2006-05-03 | Teller             |
|    10 | Paula    | Roberts   | 2006-07-27 | Head Teller        |
|    12 | Samantha | Jameson   | 2007-01-08 | Teller             |
|    13 | John     | Blake     | 2004-05-11 | Head Teller        |
|    14 | Cindy    | Mason     | 2006-08-09 | Teller             |
|    15 | Frank    | Portman   | 2007-04-01 | Teller             |
|    16 | Theresa  | Markham   | 2005-03-15 | Head Teller        |
|    17 | Beth     | Fowler    | 2006-06-29 | Teller             |
|    18 | Rick     | Tulman    | 2006-12-12 | Teller             |
+-------+----------+-----------+------------+--------------------+
15 rows in set (0.00 sec)
```

观察该输出，将会发现结果集中包括了所有 4 个 head teller 以及其他所有在 2006 年 1 月 1 日之后加入银行的雇员。employee 表中的 18 行数据中有 15 行至少满足了这两个条件中的一个。因此，当使用 and 操作符分隔条件时，在结果集中的行需要满足所有

的条件为 true；而使用 or 时，只要满足其中一个条件为 true，该行就可以被包含进来。

那么，如果需要在 where 子句中同时使用 and 和 or 操作符该怎么办呢？这就需要使用圆括号来对条件分组。下面的查询指定在结果集中返回在 2006 年 1 月 1 日之后加入公司的 head teller 雇员或者在 2007 年 1 月 1 日后入职的 teller 雇员：

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE (title = 'Head Teller' AND start_date > '2006-01-01')
    ->   OR (title = 'Teller' AND start_date > '2007-01-01');
+-------+----------+---------+------------+-------------+
|emp_id | fname    | lname   | start_date | title       |
+-------+----------+---------+------------+-------------+
|     6 | Helen    | Fleming | 2008-03-17 | Head Teller |
|     7 | Chris    | Tucker  | 2008-09-15 | Teller      |
|    10 | Paula    | Roberts | 2006-07-27 | Head Teller |
|    12 | Samantha | Jameson | 2007-01-08 | Teller      |
|    15 | Frank    | Portman | 2007-04-01 | Teller      |
+-------+----------+---------+------------+-------------+
5 rows in set (0.00 sec)
```

在混合使用不同的操作符时，开发者应当总是使用圆括号来分隔成组的条件，这样确保开发者、数据库服务器以及在以后可能会修改代码的其他开发者都能够对其意义保持一致的理解。

3.6　group by 和 having 子句

前面的查询都是仅仅提取数据而未对它们进行任何加工。不过有时候也会需要数据库服务器在返回结果集之前对数据进行一下提炼，其中的一种方式是使用 group by 子句，它用于根据列值对数据进行分组。举例来说，也许你并不需要查看雇员和他们所处部门的列表，而是想要获取部门和它所拥有雇员数的清单。在使用 group by 子句时，可能还需要同时使用 having 子句，它能够以与 where 子句类似的方式对分组数据进行过滤。

下面的查询首先为每个部门计算其所含的雇员数，然后返回至少包含 2 个雇员的部门名称：

```
mysql> SELECT d.name, count(e.emp_id) num_employees
    -> FROM department d INNER JOIN employee e
    ->   ON d.dept_id = e.dept_id
    -> GROUP BY d.name
    -> HAVING count(e.emp_id) > 2;
+----------------+---------------+
| name           |num_employees  |
+----------------+---------------+
| Administration |          3    |
```

```
| Operations     |          14   |
+----------------+---------------+
2 rows in set (0.00 sec)
```

此处只是简略地提一下这两个子句，以便读者在后面看到它们时不至于感到陌生，实际上它们相对于其他 4 个子句更为复杂一些，本书第 8 章对使用 group by 和 having 的场合与方式进行了完整的讨论。

3.7 order by 子句

通常情况下，查询的结果集返回的行并不以特定的顺序排列。如果需要对它们排序，则可以使用 order by 子句：

order by 子句用于对结果集中的原始列数据或是根据列数据计算的表达式结果进行排序。

举个例子，下面是对 account 表的一个查询：

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account;
+-----------+----------+
|open_emp_id |product_cd|
+-----------+----------+
|         10 | CHK      |
|         10 | SAV      |
|         10 | CD       |
|         10 | CHK      |
|         10 | SAV      |
|         13 | CHK      |
|         13 | MM       |
|          1 | CHK      |
|          1 | SAV      |
|          1 | MM       |
|         16 | CHK      |
|          1 | CHK      |
|          1 | CD       |
|         10 | CD       |
|         16 | CHK      |
|         16 | SAV      |
|          1 | CHK      |
|          1 | MM       |
|          1 | CD       |
|         16 | CHK      |
|         16 | BUS      |
|         10 | BUS      |
|         16 | CHK      |
|         13 | SBL      |
```

```
+-----------+-----------+
24 rows in set (0.00 sec)
```

如果需要分析每个雇员的数据，那么根据 open_emp_id 列进行排序是很有帮助的。下面简单地将该列加到 order by 子句后面：

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account
    -> ORDER BY open_emp_id;
+-----------+-----------+
|open_emp_id|product_cd |
+-----------+-----------+
|         1 | CHK       |
|         1 | SAV       |
|         1 | MM        |
|         1 | CHK       |
|         1 | CD        |
|         1 | CHK       |
|         1 | MM        |
|         1 | CD        |
|        10 | CHK       |
|        10 | SAV       |
|        10 | CD        |
|        10 | CHK       |
|        10 | SAV       |
|        10 | CD        |
|        10 | BUS       |
|        13 | CHK       |
|        13 | MM        |
|        13 | SBL       |
|        16 | CHK       |
|        16 | CHK       |
|        16 | SAV       |
|        16 | CHK       |
|        16 | BUS       |
|        16 | CHK       |
+-----------+-----------+
24 rows in set (0.00 sec)
```

现在可以清楚地看到每个雇员所开设的账户类型。不过，针对每个独立的雇员，按照同样的顺序显示账户类型也许会更好一些，那么可以在 order by 子句中的 open_emp_id 列后面再加上 product_cd：

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account
    -> ORDER BY open_emp_id, product_cd;
+-------------+------------+
| open_emp_id | product_cd |
+-------------+------------+
|         1   | CD         |
```

```
|          1   | CD          |
|          1   | CHK         |
|          1   | CHK         |
|          1   | CHK         |
|          1   | MM          |
|          1   | MM          |
|          1   | SAV         |
|         10   | BUS         |
|         10   | CD          |
|         10   | CD          |
|         10   | CHK         |
|         10   | CHK         |
|         10   | SAV         |
|         10   | SAV         |
|         13   | CHK         |
|         13   | MM          |
|         13   | SBL         |
|         16   | BUS         |
|         16   | CHK         |
|         16   | CHK         |
|         16   | CHK         |
|         16   | CHK         |
|         16   | SAV         |
+-------------+------------+
24 rows in set (0.00 sec)
```

现在结果集中首先根据雇员 ID 排序，然后根据账户类型排序。在 order by 子句中各列出现的顺序决定了对各列进行排序的次序。

3.7.1 升序或降序排序

在排序时，可以通过关键字 asc 和 desc 指定是升序还是降序。由于默认情况下是升序排序，因此只需要在想要降序排序时加上 desc 关键字即可。举个例子，下面的查询列出了根据可用余额排序的账户，并且余额最高的出现在上面：

```
mysql> SELECT account_id, product_cd, open_date, avail_balance
    -> FROM account
    -> ORDER BY avail_balance DESC;
+------------+------------+------------+---------------+
| account_id | product_cd | open_date  | avail_balance |
+------------+------------+------------+---------------+
|         29 | SBL        | 2004-02-22 |      50000.00 |
|         28 | CHK        | 2003-07-30 |      38552.05 |
|         24 | CHK        | 2002-09-30 |      23575.12 |
|         15 | CD         | 2004-12-28 |      10000.00 |
|         27 | BUS        | 2004-03-22 |       9345.55 |
|         22 | MM         | 2004-10-28 |       9345.55 |
|         12 | MM         | 2004-09-30 |       5487.09 |
|         17 | CD         | 2004-01-12 |       5000.00 |
```

```
|         18   | CHK          | 2001-05-23 |      3487.19  |
|          3   | CD           | 2004-06-30 |      3000.00  |
|          4   | CHK          | 2001-03-12 |      2258.02  |
|         13   | CHK          | 2004-01-27 |      2237.97  |
|          8   | MM           | 2002-12-15 |      2212.50  |
|         23   | CD           | 2004-06-30 |      1500.00  |
|          1   | CHK          | 2000-01-15 |      1057.75  |
|          7   | CHK          | 2002-11-23 |      1057.75  |
|         11   | SAV          | 2000-01-15 |       767.77  |
|         10   | CHK          | 2003-09-12 |       534.12  |
|          2   | SAV          | 2000-01-15 |       500.00  |
|         19   | SAV          | 2001-05-23 |       387.99  |
|          5   | SAV          | 2001-03-12 |       200.00  |
|         21   | CHK          | 2003-07-30 |       125.67  |
|         14   | CHK          | 2002-08-24 |       122.37  |
|         25   | BUS          | 2002-10-01 |         0.00  |
+------------+------------+------------+---------------+
24 rows in set (0.05 sec)
```

降序排序通常用于排行查询，比如“显示余额最高的 5 个账户”。MySQL 包含的 limit 子句允许对排序后的数据进行过滤，只显示其中的前 X 行。本书附录 B 对 limit 子句以及其他一些非 ANSI 标准的扩展进行了讨论。

3.7.2 根据表达式排序

使用列数据对结果集进行排序十分有用，但有时或许还需要根据一些并非存放在数据库中的，甚至可能没有在查询中出现的内容进行排序，而在 order by 子句后增加表达式可以满足这种需求。举例来说，对于 customer 表，也许你会需要根据客户的联邦个人识别号码（通常是个人的社会安全号码或是企业的公司号）的最后 3 位数字进行排序：

```
mysql> SELECT cust_id, cust_type_cd, city, state, fed_id
    -> FROM customer
    -> ORDER BY RIGHT(fed_id, 3);
+---------+--------------+------------+-------+-------------+
| cust_id | cust_type_cd | city       | state | fed_id      |
+---------+--------------+------------+-------+-------------+
|       1 | I            | Lynnfield  | MA    | 111-11-1111 |
|      10 | B            | Salem      | NH    | 04-1111111  |
|       2 | I            | Woburn     | MA    | 222-22-2222 |
|      11 | B            | Wilmington | MA    | 04-2222222  |
|       3 | I            | Quincy     | MA    | 333-33-3333 |
|      12 | B            | Salem      | NH    | 04-3333333  |
|      13 | B            | Quincy     | MA    | 04-4444444  |
|       4 | I            | Waltham    | MA    | 444-44-4444 |
|       5 | I            | Salem      | NH    | 555-55-5555 |
|       6 | I            | Waltham    | MA    | 666-66-6666 |
|       7 | I            | Wilmington | MA    | 777-77-7777 |
|       8 | I            | Salem      | NH    | 888-88-8888 |
```

```
|      9  | I             | Newton      | MA    | 999-99-9999 |
+---------+-------------+------------+-------+-------------+
13 rows in set (0.24 sec)
```

该查询使用内建函数 right()提取 fed_id 列的最后三个字符，并根据该值对返回的行排序。

3.7.3 根据数字占位符排序

如果需要根据 select 子句中的列来排序，那么可以选择使用该列位于 select 子句中的位置号来替代列名。举个例子，假设需要根据查询返回的第 2 个和第 5 个列排序，则可采用如下方法：

```
mysql> SELECT emp_id, title, start_date, fname, lname
    -> FROM employee
    -> ORDER BY 2, 5;
+--------+--------------------+------------+----------+-----------+
| emp_id | title              | start_date | fname    | lname     |
+--------+--------------------+------------+----------+-----------+
|     13 | Head Teller        | 2004-05-11 | John     | Blake     |
|      6 | Head Teller        | 2008-03-17 | Helen    | Fleming   |
|     16 | Head Teller        | 2005-03-15 | Theresa  | Markham   |
|     10 | Head Teller        | 2006-07-27 | Paula    | Roberts   |
|      5 | Loan Manager       | 2007-11-14 | John     | Gooding   |
|      4 | Operations Manager | 2006-04-24 | Susan    | Hawthorne |
|      1 | President          | 2005-06-22 | Michael  | Smith     |
|     17 | Teller             | 2006-06-29 | Beth     | Fowler    |
|      9 | Teller             | 2006-05-03 | Jane     | Grossman  |
|     12 | Teller             | 2007-01-08 | Samantha | Jameson   |
|     14 | Teller             | 2006-08-09 | Cindy    | Mason     |
|      8 | Teller             | 2006-12-02 | Sarah    | Parker    |
|     15 | Teller             | 2007-04-01 | Frank    | Portman   |
|      7 | Teller             | 2008-09-15 | Chris    | Tucker    |
|     18 | Teller             | 2006-12-12 | Rick     | Tulman    |
|     11 | Teller             | 2004-10-23 | Thomas   | Ziegler   |
|      3 | Treasurer          | 2005-02-09 | Robert   | Tyler     |
|      2 | Vice President     | 2006-09-12 | Susan    | Barker    |
+--------+--------------------+------------+----------+-----------+
18 rows in set (0.00 sec)
```

这种做法可能并不常见，因为如果在 select 子句中增加了新的列，而没有同步改变 order by 子句中的序号，就可能导致预料之外的结果。个人建议可以在单独的查询语句中使用列的顺序号，但是在写程序时应当总是使用名称来引用列。

3.8 小测验

下面的练习用于加深对 select 语句及其各子句的理解，附录 C 中提供了解答。

练习 3-1

获取所有银行雇员的 employee ID、名字（first name）和姓氏（last name），并先后根据姓氏和名字进行排序。

练习 3-2

获取所有状态为'ACTIVE'以及可用余额大于$2 500 的账户的 account ID、customer ID 和可用余额（available balance）。

练习 3-3

针对 account 表编写查询，以返回开设过账户的雇员 ID（使用 account.open_emp_id 列），并且结果集中每个独立的雇员只包含一行数据。

练习 3-4

为下面的多数据集查询语句填空（用<#>标记的地方），以获取所显示的结果。

```
mysql> SELECT p.product_cd, a.cust_id, a.avail_balance
    -> FROM product p INNER JOIN account <1>
    ->   ON p.product_cd = <2>
    -> WHERE p.<3> = 'ACCOUNT'
    -> ORDER BY <4>, <5>;
+------------+---------+---------------+
| product_cd | cust_id | avail_balance |
+------------+---------+---------------+
| CD         |       1 |      3000.00  |
| CD         |       6 |     10000.00  |
| CD         |       7 |      5000.00  |
| CD         |       9 |      1500.00  |
| CHK        |       1 |      1057.75  |
| CHK        |       2 |      2258.02  |
| CHK        |       3 |      1057.75  |
| CHK        |       4 |       534.12  |
| CHK        |       5 |      2237.97  |
| CHK        |       6 |       122.37  |
| CHK        |       8 |      3487.19  |
| CHK        |       9 |       125.67  |
| CHK        |      10 |     23575.12  |
| CHK        |      12 |     38552.05  |
| MM         |     3   |      2212.50  |
| MM         |       4 |      5487.09  |
| MM         |       9 |      9345.55  |
| SAV        |       1 |       500.00  |
| SAV        |       2 |       200.00  |
| SAV        |       4 |       767.77  |
| SAV        |       8 |       387.99  |
+------------+---------+---------------+
21 rows in set (0.09 sec)
```

第 4 章 过滤

有时候需要获取表中所有的行，例如：

- 提取表中所有的数据，用于建立新的数据仓库；
- 在为表增加一个新列后，修改表中的所有行；
- 获取消息队列表中的所有消息。

在这些情况下，查询不必排除表中任何一行，因此不需要在 SQL 语句中使用 where 子句。但大多数时候，查询所关注的只是表中所有行的一个子集。因此，所有的 SQL 数据处理语句（insert 语句除外）都包含了可选的 where 子句，其中的过滤条件限制了 SQL 语句所需要的行数。除此之外，select 语句中包含的 having 子句也可以对分组数据进行条件过滤。本章将探讨在 select、update 和 delete 语句中的 where 子句所能使用的各种类型的过滤条件，而对 having 子句中过滤条件的使用将在第 8 章中讨论。

4.1 条件评估

where 子句可能包含 1 个或多个条件，每个条件之间用操作符 and 和 or 分隔。如果多个条件只使用 and 操作符分隔，那么只有所有条件赋值都为 true 的行才可以被包含到结果集之中。下面给出了一个 where 子句：

```
WHERE title = 'Teller' AND start_date < '2007-01-01'
```

根据这两个条件，只有在 2007 年之前入职的出纳员（头衔为 Teller）才会出现在结果集中（换句话说，那些不是出纳员或者在 2007 年及以后入职的雇员都被排除在外了）。本例中只使用了两个条件，但实际上无论 where 子句中包含多少条件，只要它们是使用 and 操作符分隔的，就必须是所有条件都为 true 时，相应的行才可以被包含到结果集中。

如果 where 子句中的所有条件是用 or 操作符分隔的，那么只要其中一个条件成立，相应行就可以被包含到结果集中，考虑下面两个条件：

```
WHERE title = 'Teller' OR start_date < '2007-01-01'
```

现在结果集中包含行的方式发生了改变：

- 雇员是出纳员并且在 2007 年之前入职；
- 雇员是出纳员且在 2007 年 1 月 1 日以后入职；
- 雇员不是出纳员，但是是在 2007 年之前入职。

表 4-1 显示了对于 where 子句，使用 or 操作符分隔的两个条件所有可能的结果。

表 4-1　使用 or 的两条件评估

中 间 结 果	最 终 结 果
WHERE true OR true	True
WHERE true OR false	True
WHERE false OR true	True
WHERE false OR false	False

在前一个例子中，只有既不是出纳员又是在 2007 年 1 月 1 日之后入职的雇员才会被排除在结果集之外。

4.1.1　使用圆括号

如果 where 子句包含了 3 个或更多条件，且同时使用了 and 和 or 操作符，那么需要使用圆括号来明确意图，以使数据库服务器或者以后可能阅读你代码的其他开发者能够理解。下面的 where 子句是对前一个例子的扩展，以检查该雇员是否仍然被银行雇佣：

```
WHERE end_date IS NULL
  AND (title = 'Teller' OR start_date < '2007-01-01')
```

其中包含了 3 个条件，用于检验数据行是否应当被加入到结果集。第一个条件值必须为 true，而第二个和第三个条件值只需要有一个（或者两个都）为 true。表 4-2 显示了这 3 个条件所有可能的结果。

表 4-2　使用 and、or 的三条件评估

中 间 结 果	最 终 结 果
WHERE true AND (true OR true)	True
WHERE true AND (true OR false)	True
WHERE true AND (false OR true)	True
WHERE true AND (false OR false)	False

续表

中 间 结 果	最 终 结 果
WHERE false AND (true OR true)	False
WHERE false AND (true OR false)	False
WHERE false AND (false OR true)	False
WHERE false AND (false OR false)	False

如上表所示，where 子句中所包含的条件越多，服务器所需要评估的中间结果也越多。在本例中，8 个中间结果中的 3 个最终结果为 true。

4.1.2 使用 not 操作符

希望前面包含 3 个条件的例子能够易于被读者理解，继续考虑下面的修改：

```
WHERE end_date IS NULL
  AND NOT (title = 'Teller' OR start_date < '2007-01-01')
```

注意到与前一个例子的差异在哪里吗？本例在第二行的 and 操作符后面增加了 not 操作符。现在不再是查找身为出纳员或者在 2007 年之前入职的在职雇员了，而是查找既不是出纳员又在 2007 年或之后入职的在职雇员。表 4-3 中显示了本例可能的结果。

表 4-3 使用 and、or 和 not 的三条件评估

中 间 结 果	最 终 结 果
WHERE true AND NOT (true OR true)	False
WHERE true AND NOT (true OR false)	False
WHERE true AND NOT (false OR true)	False
WHERE true AND NOT (false OR false)	True
WHERE false AND NOT (true OR true)	False
WHERE false AND NOT (true OR false)	False
WHERE false AND NOT (false OR true)	False
WHERE false AND NOT (false OR false)	False

尽管对于数据库服务器来说，处理包含 not 操作符的 where 子句毫不费力，但对于开发者来说，增加了对条件评估的困难，这也是通常情况下较少使用它的原因。本例中可以重写 where 子句，以避免使用 not 操作符：

```
WHERE end_date IS NULL
  AND title != 'Teller' AND start_date >= '2007-01-01'
```

我相信，尽管对于服务器来说这个版本的 where 子句并没有差别，但对于开发者来说，

它变得更易于理解了。

4.2　构建条件

上面已经介绍了服务器是如何对多个条件进行评估的，这里将会回过头来讨论如何创建单个条件。条件通常由 1 个或多个包含 1 个到多个操作符的表达式构成。表达式可以是下面类型中的任意一个：

- 数字；
- 表或视图中的列；
- 字符串，比如'Teller'；
- 内建函数，比如函数 concat('Learning', ' ', 'SQL')；
- 子查询；
- 表达式列表（如'Teller', 'Head Teller', 'Operations Manager'）；
- 比较操作符，如=、!=、<、>、<>、LIKE、IN 和 BETWEEN；
- 算术操作符，比如 +、 −、 *、和 /。

下面一节将讲述如何联合使用这些表达式和操作符，以产生不同类型的条件。

4.3　条件类型

有许多种方式可以过滤掉不需要的数据，比如通过指定特定值、值集合、需要包含或排除的值的范围，以及在针对字符串数据时，使用各种模式搜索技术来查找部分匹配。下面将详细介绍这些条件类型。

4.3.1　相等条件

你所编写或遇到的相当一部分过滤条件类似于‘column=expression’的形式，例如：

```
title = 'Teller'
fed_id = '111-11-1111'
amount = 375.25
dept_id = (SELECT dept_id FROM department WHERE name = 'Loans')
```

这些条件被称为相等条件，因为它们将一个表达式等同于另一个表达式。前 3 个例子是将列等同于符号（两个字符串和 1 个数字），第 4 个例子将列等同于子查询的返回值。下面的查询使用两个相等条件：其中一个在 on 子句中（连接条件），另一个在 where 子句中（过滤条件）：

```
mysql> SELECT pt.name product_type, p.name product
    -> FROM product p INNER JOIN product_type pt
    ->   ON p.product_type_cd = pt.product_type_cd
    -> WHERE pt.name = 'Customer Accounts';
+-------------------+------------------------+
| product_type      | product                |
+-------------------+------------------------+
| Customer Accounts | certificate of deposit |
| Customer Accounts | checking account       |
| Customer Accounts | money market account   |
| Customer Accounts | savings account        |
+-------------------+------------------------+
4 rows in set (0.08 sec)
```

该查询显示了所有类型为 customer account 的产品。

不等条件

另一种比较常见的条件类型是不等条件，它用于判断两个表达式不相等。下面的查询将前面例子中 where 子句的过滤条件改为不等条件：

```
mysql> SELECT pt.name product_type, p.name product
    -> FROM product p INNER JOIN product_type pt
    ->   ON p.product_type_cd = pt.product_type_cd
    -> WHERE pt.name <> 'Customer Accounts';
+-------------------------------+-------------------------+
| product_type                  | product                 |
+-------------------------------+-------------------------+
| Individual and Business Loans | auto loan               |
| Individual and Business Loans | business line of credit |
| Individual and Business Loans | home mortgage           |
| Individual and Business Loans | small business loan     |
+-------------------------------+-------------------------+
4 rows in set (0.00 sec)
```

该查询显示了所有不是 customer account 类型的账户。在构造不等条件时，可以在!=或<>操作符中任选一个。

使用相等条件修改数据

在修改数据时经常要用到相等/不等条件。举个例子，假设银行每年需要删除过期的账户数据行，具体来说，需要删除 account 表中在 2002 年关闭的账户行，下面提供一种实现方式：

```
DELETE FROM account
WHERE status = 'CLOSED' AND YEAR(close_date) = 2002;
```

该语句包含了两个相等条件：一个用于限定只查找关闭账户，另一个检查这些账户是否于 2002 年关闭。

提示

在使用 delete 或 update 语句的示例时，本书尽量保证实际上并没有数据行被修改。这样在执行完这些语句后，数据仍能保持无变化，从而保证了后面例子中 select 语句的输出始终与本书所显示的相一致。

由于 MySQL 中的会话被默认设置为自动提交模式（参见第 12 章），因此如果有语句对数据进行了修改，那么将无法对示例数据的变动进行回滚（撤销）。当然你可以随意操作你的示例数据，甚至可以完全清除它们然后再次运行前面提供的脚本，但我建议尽量保持数据的完整性。

4.3.2 范围条件

除了可以检查表达式与另一个表达式是否相等以外，还可以构建条件来检验表达式的值是否处于某个区间。这种类型的条件通常用于数值型或临时数据，考虑下面的查询：

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date < '2007-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
|      1 | Michael | Smith     | 2005-06-22 |
|      2 | Susan   | Barker    | 2006-09-12 |
|      3 | Robert  | Tyler     | 2005-02-09 |
|      4 | Susan   | Hawthorne | 2006-04-24 |
|      8 | Sarah   | Parker    | 2006-12-02 |
|      9 | Jane    | Grossman  | 2006-05-03 |
|     10 | Paula   | Roberts   | 2006-07-27 |
|     11 | Thomas  | Ziegler   | 2004-10-23 |
|     13 | John    | Blake     | 2004-05-11 |
|     14 | Cindy   | Mason     | 2006-08-09 |
|     16 | Theresa | Markham   | 2005-03-15 |
|     17 | Beth    | Fowler    | 2006-06-29 |
|     18 | Rick    | Tulman    | 2006-12-12 |
+--------+---------+-----------+------------+
13 rows in set (0.15 sec)
```

该查询找出所有 2007 年之前雇佣的职员，但除了指定开始日期的上限外，或许还需要指定开始日期的下限。

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date < '2007-01-01'
    ->   AND start_date >= '2005-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
```

```
|      1 | Michael | Smith     | 2005-06-22 |
|      2 | Susan   | Barker    | 2006-09-12 |
|      3 | Robert  | Tyler     | 2005-02-09 |
|      4 | Susan   | Hawthorne | 2006-04-24 |
|      8 | Sarah   | Parker    | 2006-12-02 |
|      9 | Jane    | Grossman  | 2006-05-03 |
|     10 | Paula   | Roberts   | 2006-07-27 |
|     14 | Cindy   | Mason     | 2006-08-09 |
|     16 | Theresa | Markham   | 2005-03-15 |
|     17 | Beth    | Fowler    | 2006-06-29 |
|     18 | Rick    | Tulman    | 2006-12-12 |
+--------+---------+-----------+------------+
11 rows in set (0.00 sec)
```

此查询获取在 2005 年或 2006 年雇佣的职员。

between 操作符

当需要同时限制范围的上限和下限时，可以选择使用 between 操作符构建一个查询条件，而不需要两个单独的限制条件：

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date BETWEEN '2005-01-01' AND '2007-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
|      1 | Michael | Smith     | 2005-06-22 |
|      2 | Susan   | Barker    | 2006-09-12 |
|      3 | Robert  | Tyler     | 2005-02-09 |
|      4 | Susan   | Hawthorne | 2006-04-24 |
|      8 | Sarah   | Parker    | 2006-12-02 |
|      9 | Jane    | Grossman  | 2006-05-03 |
|     10 | Paula   | Roberts   | 2006-07-27 |
|     14 | Cindy   | Mason     | 2006-08-09 |
|     16 | Theresa | Markham   | 2005-03-15 |
|     17 | Beth    | Fowler    | 2006-06-29 |
|     18 | Rick    | Tulman    | 2006-12-12 |
+--------+---------+-----------+------------+
11 rows in set (0.03 sec)
```

当使用 between 操作符时，需要注意下面的事项，即必须首先指定范围的下限（在 between 后面），然后指定范围的上限（在 end 的后面）。如果错误地指定了它们出现的次序，则会产生下面的结果：

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date BETWEEN '2007-01-01' AND '2005-01-01';
Empty set (0.00 sec)
```

如上所示，结果没有返回任何数据，这是因为服务端实际上根据该条件产生了两个使用<=和>=操作符的条件：

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date >= '2007-01-01'
    ->   AND start_date <= '2005-01-01';
Empty set (0.00 sec)
```

因为不可能存在大于 2007 年 1 月 1 日且小于 2005 年 1 月 1 日的日期，所以查询只能返回空的结果集。这里暗示了使用 between 所需要注意的第二个事项，即范围的上下限是闭合的，也就是说上下限值本身也被包含在范围内。因此在本例中，最好使用 2005-01-01 和 2006-12-31 分别作为日期范围的起点和终点，而不是使用 2007-01-01。尽管实际上在 2007 年元旦当天银行很可能不会有新职员加入，但是这更精确地体现了此查询的需求。

除了日期，还可以根据数字范围来构造条件，这种方式更易于掌握，例如：

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE avail_balance BETWEEN 3000 AND 5000;
+------------+-----------+---------+--------------+
| account_id | product_cd | cust_id |avail_balance  |
+------------+-----------+---------+--------------+
|        3   |   CD       |       1 |       3000.00 |
|       17   |   CD       |       7 |       5000.00 |
|       18   |   CHK      |       8 |       3487.19 |
+------------+-----------+---------+--------------+
3 rows in set (0.10 sec)
```

该查询返回了所有余额在$3000 和$5000 之间的账户信息。同样地，这里需要先指定范围的下限。

字符串范围

使用日期或数字的范围是易于理解的，但同样可以使用字符串作为搜索范围的条件，当然其结果不是显而易见的。举一个查询客户的社会安全号码的例子，社会安全号码的格式为“XXX-XX-XXXX”，其中 X 为 0 到 9 之间的数字。当需要查询所有社会安全号码位于“500-00-0000”和“999-99-9999”之间的客户时，可以使用下面的语句：

```
mysql> SELECT cust_id, fed_id
    -> FROM customer
    -> WHERE cust_type_cd = 'I'
    ->   AND fed_id BETWEEN '500-00-0000' AND '999-99-9999';
+---------+------------+
|cust_id  |  fed_id     |
+---------+------------+
|     5   | 555-55-5555 |
```

```
|     6   | 666-66-6666 |
|     7   | 777-77-7777 |
|     8   | 888-88-8888 |
|     9   | 999-99-9999 |
+---------+-------------+
5 rows in set (0.01 sec)
```

使用字符串范围时，需要知道所使用的字符集中各字符的次序（在某个字符集内各字符的次序被称为排序顺序（collation））。

4.3.3　成员条件

在一些情况下，不是需要限制表达式为特定值或某个范围的值，而是一个有限值集合。举例来说，在 account 表中找出所有产品代码为‘CHK’、‘SAV’、‘CD’或‘MM’的账户：

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd = 'CHK' OR product_cd = 'SAV'
    ->   OR product_cd = 'CD' OR product_cd = 'MM';
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         1  | CHK        |     1   |      1057.75  |
|         2  | SAV        |     1   |       500.00  |
|         3  | CD         |     1   |      3000.00  |
|         4  | CHK        |     2   |      2258.02  |
|         5  | SAV        |     2   |       200.00  |
|         7  | CHK        |     3   |      1057.75  |
|         8  | MM         |     3   |      2212.50  |
|        10  | CHK        |     4   |       534.12  |
|        11  | SAV        |     4   |       767.77  |
|        12  | MM         |     4   |      5487.09  |
|        13  | CHK        |     5   |      2237.97  |
|        14  | CHK        |     6   |       122.37  |
|        15  | CD         |     6   |     10000.00  |
|        17  | CD         |     7   |      5000.00  |
|        18  | CHK        |     8   |      3487.19  |
|        19  | SAV        |     8   |       387.99  |
|        21  | CHK        |     9   |       125.67  |
|        22  | MM         |     9   |      9345.55  |
|        23  | CD         |     9   |      1500.00  |
|        24  | CHK        |    10   |     23575.12  |
|        28  | CHK        |    12   |     38552.05  |
+------------+------------+---------+---------------+
21 rows in set (0.28 sec)
```

本例中的 where 子句（一共包含了 4 个条件）也许没有复杂到难以编写的程度，但想像一下如果表达式集合中包含了 10 个甚至 20 个成员就令人畏惧了。对于这些情况，

可以使用 in 操作符：

```
SELECT account_id, product_cd, cust_id, avail_balance
FROM account
WHERE product_cd IN ('CHK','SAV','CD','MM');
```

无论集合中含有多少表达式，使用 in 操作符都只需要编写一个条件。

使用子查询

除了编写自定义的表达式集合，如('CHK','SAV','CD','MM')之外，还可以使用子查询产生中间集合。举个例子，在前一个查询中，所有 4 种产品类型的 product_type_cd 列都为‘ACCOUNT’，因此可以使用对 product 表的子查询来获取这 4 种产品代码，而不是显式地列举它们：

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd IN (SELECT product_cd FROM product
    ->  WHERE product_type_cd = 'ACCOUNT');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|          3 | CD         |       1 |       3000.00 |
|         15 | CD         |       6 |      10000.00 |
|         17 | CD         |       7 |       5000.00 |
|         23 | CD         |       9 |       1500.00 |
|          1 | CHK        |       1 |       1057.75 |
|          4 | CHK        |       2 |       2258.02 |
|          7 | CHK        |       3 |       1057.75 |
|         10 | CHK        |       4 |        534.12 |
|         13 | CHK        |       5 |       2237.97 |
|         14 | CHK        |       6 |        122.37 |
|         18 | CHK        |       8 |       3487.19 |
|         21 | CHK        |       9 |        125.67 |
|         24 | CHK        |      10 |      23575.12 |
|         28 | CHK        |      12 |      38552.05 |
|          8 | MM         |       3 |       2212.50 |
|         12 | MM         |       4 |       5487.09 |
|         22 | MM         |       9 |       9345.55 |
|          2 | SAV        |       1 |        500.00 |
|          5 | SAV        |       2 |        200.00 |
|         11 | SAV        |       4 |        767.77 |
|         19 | SAV        |       8 |        387.99 |
+------------+------------+---------+---------------+
21 rows in set (0.11 sec)
```

子查询返回了包含 4 个值的集合，主查询检查每个产品 product_cd 列的值是否属于子查询所返回的集合。

使用 not in

有时候需要检查一个表达式是否在某个表达式集合中存在，但有时候需要检查的是它是否不存在。对于这些情况，可以使用 not in 操作符：

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd NOT IN ('CHK','SAV','CD','MM');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         25 | BUS        |      10 |          0.00 |
|         27 | BUS        |      11 |       9345.55 |
|         29 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
3 rows in set (0.09 sec)
```

该查询查找所有不是 checking、saving、certificate of deposit 和 money market accounts 的账户。

4.3.4　匹配条件

到此为止，本章已经介绍了识别单个字符串、字符串范围或字符串集合的条件，而最后一种条件类型是处理部分字符串匹配。举例来说，或许你需要找到所有具有以 T 开头的姓氏的雇员，那么可以使用内建函数截取 lname 列中的第一个字母，如下所示。

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE LEFT(lname, 1) = 'T';
+--------+--------+--------+
| emp_id | fname  | lname  |
+--------+--------+--------+
|      3 | Robert | Tyler  |
|      7 | Chris  | Tucker |
|     18 | Rick   | Tulman |
+--------+--------+--------+
3 rows in set (0.01 sec)
```

尽管内建函数 left()发挥了作用，但它并不具备较好的灵活性。因此使用通配符构建搜索表达式是更好的做法，如下面所述。

使用通配符

当根据部分字符串匹配进行搜索时，感兴趣的项目可能包括：

- 以某个字符开始（或结束）的字符串；
- 包含某个子字符串的字符串；
- 在字符串的任意位置包含某个字符的字符串；

- 在字符串的任意位置包含某个子字符串的字符串；
- 具备特定格式而不关心单个字符的字符串。

可以构建搜索表达式定位这些字符串，表 4-4 中列举了更多的使用通配符的部分字符串匹配方式。

表 4-4 通配符

通 配 符	匹 配
_	正好 1 个字符
%	任意数目的字符（包括 0）

下划线为单个字符的占位符，百分号则代表多个字符。当使用搜索表达式构建条件时，可以使用 like 操作符，如下所示。

```
mysql> SELECT lname
    -> FROM employee
    -> WHERE lname LIKE '_a%e%';
+-----------+
| lname     |
+-----------+
| Barker    |
| Hawthorne |
| Parker    |
| Jameson   |
+-----------+
4 rows in set (0.00 sec)
```

上例中的搜索表达式指定字符串的第二个位置必须为字符 a， e 必须在其后的任何位置（包括最后一个位置）中至少出现一次。表 4-5 展示了更多的搜索表达式及相应解释。

表 4-5 搜索表达式示例

搜索表达式	解 释
F%	以 F 打头的字符串
%t	以 t 结尾的字符串
%bas%	包含‘bas’子字符串的字符串
_ _t_	包含 4 个字符且第 3 个字符为 t 的字符串
_ _ _-_ _-_ _ _ _	包含 11 个字符且第 4 和第 7 个字符为破折号的字符串

可以使用表 4-5 中的最后一个示例来查找所有联邦个人识别号码与社会安全号码的格式相匹配的顾客，如下所示：

```
mysql> SELECT cust_id, fed_id
    -> FROM customer
    -> WHERE fed_id LIKE '___-__-____';
+---------+------------+
| cust_id | fed_id     |
+---------+------------+
|       1 |111-11-1111 |
|       2 |222-22-2222 |
|       3 |333-33-3333 |
|       4 |444-44-4444 |
|       5 |555-55-5555 |
|       6 |666-66-6666 |
|       7 |777-77-7777 |
|       8 |888-88-8888 |
|       9 |999-99-9999 |
+---------+------------+
9 rows in set (0.02 sec)
```

对于构建简单的搜索表达式，使用通配符是非常方便的。如果需要处理更为复杂的情况，可以使用多个搜索表达式，例如：

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE lname LIKE 'F%' OR lname LIKE 'G%';
+--------+-------+----------+
| emp_id | fname | lname    |
+--------+-------+----------+
|      5 | John  | Gooding  |
|      6 | Helen | Fleming  |
|      9 | Jane  | Grossman |
|     17 | Beth  | Fowler   |
+--------+-------+----------+
4 rows in set (0.00 sec)
```

该查询查找所有姓氏以 F 或 G 打头的雇员。

使用正则表达式

如果带通配符的字符串仍然不能提供足够的灵活性，那么可以使用正则表达式来构造搜索表达式。正则表达式实质上也是一种特殊的搜索表达式，有的读者虽然不熟悉 SQL，但曾使用过 Perl 等脚本编程语言，那么可能已经对正则表达式十分熟悉了。如果你从未使用过正则表达式，可以参考 Jeffrey E.F. Friedl 的 *Mastering Regular Expression* 一书。因为它是个非常大的主题，所以本书中无法详细描述。

下面给出前一个查询（查询姓以字母 F 或 G 开头的职员）在 MySQL 中的正则表达式实现：

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
```

```
    -> WHERE lname REGEXP '^[FG]';
+--------+-------+----------+
| emp_id | fname | lname    |
+--------+-------+----------+
|      5 | John  | Gooding  |
|      6 | Helen | Fleming  |
|      9 | Jane  | Grossman |
|     17 | Beth  | Fowler   |
+--------+-------+----------+
4 rows in set (0.00 sec)
```

regexp 操作符接受一个正则表达式（本例中为'^[FG]'），并将之应用到操作符的左侧表达式（本例为 lname 列）。现在该查询只包含一个使用正则表达式的条件，而非两个使用通配符的条件。

Oracle 数据库和 Microsoft SQL Server 同样支持正则表达式。Oracle 中使用 regexp_like 函数替代前面例子中 regexp 操作符的作用，而 SQL Server 则允许在 like 操作符中使用正则表达式。

4.4 null: 4 个字母的关键字

尽管我尽量推迟，但现在还是到了要讨论令人感到模糊和害怕的内容——null 值的时候了。null 表示值的缺失，举例来说，在职员离职之前，employee 表的 end_date 列没有可赋予的值，因此它应该一直为 null。当然 null 的使用方式比较灵活，下面是其不同的适用场合：

没有合适的值

比如 ATM 机上的自助交易并不需要 employee ID 列；

值未确定

比如在客户行被创建时还不知道他的 federal ID；

值未定义

比如为某个还未添加到数据库的产品创建账户。

提示

一些理论学者认为针对上面（以及更多）的每种情况，都应该提供不同的表达式，但大多数实践专家都认为使用多种 null 值的表示方法会带来更多的困扰。

当使用 null 时，需要记住：

- 表达式可以为 null，但不能等于 null；

- 两个 null 值彼此不能判断为相等。

为了测试表达式是否为 null，需要使用 null 操作符，如下所示。

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id IS NULL;
+--------+---------+-------+-----------------+
| emp_id | fname   | lname | superior_emp_id |
+--------+---------+-------+-----------------+
|     1  | Michael | Smith |          NULL   |
+--------+---------+-------+-----------------+
1 row in set (0.00 sec)
```

该查询返回所有没有主管的雇员（是不是美好的梦想）。下面的查询使用= null 替换 is null：

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id = NULL;
Empty set (0.01 sec)
```

如结果所示，该查询虽然被解析和执行，但并没有返回任何数据行。这是不熟练的 SQL 程序员常犯的错误，并且数据库服务器不会为该错误产生告警信息，因此在构建测试空值的查询时必须特别小心。

如果想要检查列中数据是否被赋值，可以使用 not null 操作符，如下所示。

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id IS NOT NULL;
+--------+----------+-----------+---------------+
| emp_id | fname    | lname     |superior_emp_id |
+--------+----------+-----------+---------------+
|     2  | Susan    | Barker    |             1  |
|     3  | Robert   | Tyler     |             1  |
|     4  | Susan    | Hawthorne |             3  |
|     5  | John     | Gooding   |             4  |
|     6  | Helen    | Fleming   |             4  |
|     7  | Chris    | Tucker    |             6  |
|     8  | Sarah    | Parker    |             6  |
|     9  | Jane     | Grossman  |             6  |
|    10  | Paula    | Roberts   |             4  |
|    11  | Thomas   | Ziegler   |            10  |
|    12  | Samantha | Jameson   |            10  |
|    13  | John     | Blake     |             4  |
|    14  | Cindy    | Mason     |            13  |
|    15  | Frank    | Portman   |            13  |
|    16  | Theresa  | Markham   |             4  |
|    17  | Beth     | Fowler    |            16  |
```

```
|     18 | Rick     | Tulman     |              16 |
+--------+----------+-----------+-----------------+
17 rows in set (0.00 sec)
```

该版本的查询返回了除了 Michael Smith 之外的 17 名雇员，他们都有自己的主管。

下面暂时把 null 放到一边，介绍另一种易犯的错误，这可能更有帮助。假设你需要查找所有不是 Helen Fleming（employee ID 为 6）所管理的雇员，那么脑中第一时间出现的做法可能是：

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id != 6;
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      2 | Susan    | Barker    |            1    |
|      3 | Robert   | Tyler     |            1    |
|      4 | Susan    | Hawthorne |            3    |
|      5 | John     | Gooding   |            4    |
|      6 | Helen    | Fleming   |            4    |
|     10 | Paula    | Roberts   |            4    |
|     11 | Thomas   | Ziegler   |           10    |
|     12 | Samantha | Jameson   |           10    |
|     13 | John     | Blake     |            4    |
|     14 | Cindy    | Mason     |           13    |
|     15 | Frank    | Portman   |           13    |
|     16 | Theresa  | Markham   |            4    |
|     17 | Beth     | Fowler    |           16    |
|     18 | Rick     | Tulman    |           16    |
+--------+----------+-----------+-----------------+
14 rows in set (0.00 sec)
```

尽管返回的 14 个雇员的确不为 Helen Fleming 工作，但仔细观察数据，就会发现还有一个不为 Helen 工作的雇员并没有列出来。该雇员为 Michael Smith，他的 superior_emp_id 列为 null（因为他是银行的高层人物）。因此为了获取正确的结果，需要检查所有 superior_emp_id 列为空的数据行。

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id != 6 OR superior_emp_id IS NULL;
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      1 | Michael  | Smith     |         NULL    |
|      2 | Susan    | Barker    |            1    |
|      3 | Robert   | Tyler     |            1    |
|      4 | Susan    | Hawthorne |            3    |
|      5 | John     | Gooding   |            4    |
```

```
|       6 | Helen    | Fleming   |              4 |
|      10 | Paula    | Roberts   |              4 |
|      11 | Thomas   | Ziegler   |             10 |
|      12 | Samantha | Jameson   |             10 |
|      13 | John     | Blake     |              4 |
|      14 | Cindy    | Mason     |             13 |
|      15 | Frank    | Portman   |             13 |
|      16 | Theresa  | Markham   |              4 |
|      17 | Beth     | Fowler    |             16 |
|      18 | Rick     | Tulman    |             16 |
+--------+----------+-----------+----------------+
15 rows in set (0.00 sec)
```

现在结果集包含了所有 15 名不为 Helen 工作的雇员。当使用不熟悉的数据库时，好的做法是首先确定表中哪些列可以允许 null 值，以便在过滤条件中采取适当的措施确保不会漏掉所需要的数据。

4.5 小测验

下面的练习测试读者对过滤条件的理解程度，附录 C 提供了练习答案。

前两个练习所用到的交易数据如下。

Txn_id	Txn_date	Account_id	Txn_type_cd	Amount
1	2005-02-22	101	CDT	1000.00
2	2005-02-23	102	CDT	525.75
3	2005-02-24	101	DBT	100.00
4	2005-02-24	103	CDT	55
5	2005-02-25	101	DBT	50
6	2005-02-25	103	DBT	25
7	2005-02-25	102	CDT	125.37
8	2005-02-26	103	DBT	10
9	2005-02-27	101	CDT	75

练习 4-1

下面的过滤条件将返回哪些交易的 ID？

```
txn_date < '2005-02-26' AND (txn_type_cd = 'DBT' OR amount > 100)
```

练习 4-2

下面的过滤条件将返回哪些交易的 ID？

```
account_id IN (101,103) AND NOT (txn_type_cd = 'DBT' OR amount > 100)
```

练习 4-3

构造查询语句，获取在 2002 年开户的所有账户。

练习 4-4

构造查询,查找姓氏中以 a 为第二个字符,并且 e 在 a 后面任意位置出现的非公司顾客。

第 5 章

多表查询

在第 2 章中，介绍了如何通过正交化过程将相关概念分割成若干独立的部分，在示例中产生的最终结果为两个表：person 和 favorite_food。但是，如果需要创建一个显示某个人姓名、地址和所喜欢食物的报表，就需要某种机制将这两张表中的数据再次整合到一起，这种机制被称为连接（join）。本章关注最简单也是最常用的连接，即内连接（inner join）。第 10 章讨论了其他所有的连接类型。

5.1 什么是连接

尽管对单个表的查询并不罕见，但在应用环境下大多数的查询都需要针对两个、3 个甚至更多的表。下面举例来说明，首先查看 employee 和 department 表的定义，然后定义一个需要从这两个表获取数据的查询：

```
mysql> DESC employee;
+--------------------+--------------------+------+-----+---------+
| Field              | Type               | Null | Key | Default |
+--------------------+--------------------+------+-----+---------+
| emp_id             |smallint(5)unsigned | NO   | PRI | NULL    |
| fname              |varchar(20)         | NO   |     | NULL    |
| lname              |varchar(20)         | NO   |     | NULL    |
| start_date         |date                | NO   |     | NULL    |
| end_date           |date                | YES  |     | NULL    |
| superior_emp_id    |smallint(5)unsigned | YES  | MUL | NULL    |
| dept_id            |smallint(5)unsigned | YES  | MUL | NULL    |
| title              |varchar(20)         | YES  |     | NULL    |
| assigned_branch_id |smallint(5)unsigned | YES  | MUL | NULL    |
+--------------------+--------------------+------+-----+---------+
9 rows in set (0.11 sec)
mysql> DESC department;
+-------------------+-------------------+-------+-----+--------+
```

```
| Field   | Type                 | Null   | Key   | Default      |
+---------+----------------------+--------+-------+--------------+
| dept_id | smallint(5) unsigned | No     | PRI   | NULL         |
| name    | varchar(20)          | No     |       | NULL         |
+---------+----------------------+--------+-------+--------------+
2 rows in set (0.03 sec)
```

假设现在你需要获取每个雇员的姓名以及他们所在的部门名称，显然此查询需要获取 employee.fname、employee.lname 和 department.name 列。但是如何在同一查询中获取两个表的数据呢？答案的关键在于 employee.dept_id 列，它保存了每个雇员所在的部门 ID（更正式的说法是，employee.dept_id 列是指向 department 表的外键）。该查询将指示服务器使用 employee.dept_id 列作为 employee 和 department 表的桥梁，从而实现在同一查询的结果集中包含来自两个表的列，这种操作被称为连接。

5.1.1　笛卡儿积

最简单的连接方式是直接在 from 子句中加入 employee 表和 department 表。下面的查询获取雇员的姓名以及部门名，在 from 子句中包含了两个表，并使用关键字 join 隔开：

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e JOIN department d;
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Michael  | Smith     | Operations     |
| Michael  | Smith     | Loans          |
| Michael  | Smith     | Administration |
| Susan    | Barker    | Operations     |
| Susan    | Barker    | Loans          |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Operations     |
| Robert   | Tyler     | Loans          |
| Robert   | Tyler     | Administration |
| Susan    | Hawthorne | Operations     |
| Susan    | Hawthorne | Loans          |
| Susan    | Hawthorne | Administration |
| John     | Gooding   | Operations     |
| John     | Gooding   | Loans          |
| John     | Gooding   | Administration |
| Helen    | Fleming   | Operations     |
| Helen    | Fleming   | Loans          |
| Helen    | Fleming   | Administration |
| Chris    | Tucker    | Operations     |
| Chris    | Tucker    | Loans          |
| Chris    | Tucker    | Administration |
| Sarah    | Parker    | Operations     |
| Sarah    | Parker    | Loans          |
| Sarah    | Parker    | Administration |
```

```
| Jane     | Grossman | Operations     |
| Jane     | Grossman | Loans          |
| Jane     | Grossman | Administration |
| Paula     | Roberts | Operations     |
| Paula    | Roberts  | Loans          |
| Paula    | Roberts  | Administration |
| Thomas   | Ziegler  | Operations     |
| Thomas   | Ziegler  | Loans          |
| Thomas   | Ziegler  | Administration |
| Samantha | Jameson  | Operations     |
| Samantha | Jameson  | Loans          |
| Samantha | Jameson  | Administration |
| John     | Blake    | Operations     |
| John     | Blake    | Loans          |
| John     | Blake    | Administration |
| Cindy    | Mason    | Operations     |
| Cindy    | Mason    | Loans          |
| Cindy    | Mason    | Administration |
| Frank    | Portman  | Operations     |
| Frank    | Portman  | Loans          |
| Frank    | Portman  | Administration |
| Theresa  | Markham  | Operations     |
| Theresa  | Markham  | Loans          |
| Theresa  | Markham  | Administration |
| Beth     | Fowler   | Operations     |
| Beth     | Fowler   | Loans          |
| Beth     | Fowler   | Administration |
| Rick     | Tulman   | Operations     |
| Rick     | Tulman   | Loans          |
| Rick     | Tulman   | Administration |
+----------+----------+----------------+
54 rows in set (0.23 sec)
```

一共有 18 个雇员和 3 个部门，但最后的结果集为什么包含了 54 行呢？再仔细观察一下，将会发现 18 个雇员组成的集合被重复了 3 次，每次除了部门名称不同之外，其他数据项都是完全相同的。这是由于查询并没有指定两个表应如何连接造成的。在此情况下，数据库服务器将产生笛卡儿积，即两张表的所有置换（18 个雇员 × 3 个部门=54 个置换）。这种连接被称为交叉连接（cross join），它在实际应用中很少使用。交叉连接是第 10 章中所讨论的连接类型之一。

5.1.2　内连接

要修改上一个查询以使结果集中只包含 18 行（每个雇员一行），则需要描述两个表是如何关联的。前面已经提到 employee.dept_id 列起到连接这两个的作用，因此需要在 from 子句中增加一个包含此列信息的子句：

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e JOIN department d
```

```
    ->  ON e.dept_id = d.dept_id;
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
| Susan    | Hawthorne | Operations     |
| John     | Gooding   | Loans          |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
+----------+-----------+----------------+
18 rows in set (0.00 sec)
```

结果集中现在只出现了所期望的 18 行，而不是原先的 54 行，这是 on 子句产生的作用，它提示服务器使用 dept_id 列连接 employee 和 departement 表。例如，employee 表中 Susan Hawthorne 一行包含的 dept_id 列值为 1（ID 列在本例中没有显示），服务器使用该值查找 department 表中 dept_id 列为 1 的行，并获取该行的 name 列值，即‘Operations’。

如果在一个表中的 dept_id 列中存在某个值，但该值在另一张表的 dept_id 列中不存在，那么相关行的连接会失败，在结果集中将会排除包含该值的行。这种类型的连接被称为内连接，也是最常用的一种连接类型。具体来说，假设 department 表中的第 4 行为市场部门，但是 employee 表中没有雇员被赋予到此部门下，那么在连接后的结果集中将不会出现市场部门。与之对应的是，如果一些雇员被指定到 ID 为 99 的部门，但该部门在 department 表中不存在，那么这些雇员行也会被结果集排除在外。如果想要包含其中某个表的所有行，而不考虑每行是否在另一个表中存在匹配，那么可以使用外连接（outer join），本书后面将会详细介绍这方面的内容。

在前一个例子中，from 子句并没有指定所使用的连接类型。通常在对两个表使用内连接时，最好在 from 子句中显式指定连接类型，下面的例子提供同样的查询，只不过增加了连接类型（注意关键字 INNER）：

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e INNER JOIN department d
    ->  ON e.dept_id = d.dept_id;
```

```
+----------+----------+----------------+
| fname    | lname    | name           |
+----------+----------+----------------+
| Michael  | Smith    | Administration |
| Susan    | Barker   | Administration |
| Robert   | Tyler    | Administration |
| Susan    | Hawthorne| Operations     |
| John     | Gooding  | Loans          |
| Helen    | Fleming  | Operations     |
| Chris    | Tucker   | Operations     |
| Sarah    | Parker   | Operations     |
| Jane     | Grossman | Operations     |
| Paula    | Roberts  | Operations     |
| Thomas   | Ziegler  | Operations     |
| Samantha | Jameson  | Operations     |
| John     | Blake    | Operations     |
| Cindy    | Mason    | Operations     |
| Frank    | Portman  | Operations     |
| Theresa  | Markham  | Operations     |
| Beth     | Fowler   | Operations     |
| Rick     | Tulman   | Operations     |
+----------+----------+----------------+
18 rows in set (0.00 sec)
```

如果并没有指定连接类型，那么服务器将会默认使用内连接。正如本书后面所介绍的，SQL 中还存在其他几种连接类型，因此最好养成在使用连接时明确指定类型的习惯。

如果连接两个表的列名是相同的，如前一个例子中的情况，那么可以使用 using 子句替代 on 子句，如下所示。

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e INNER JOIN department d
    ->   USING (dept_id);
+----------+----------+----------------+
| fname    | lname    | name           |
+----------+----------+----------------+
| Michael  | Smith    | Administration |
| Susan    | Barker   | Administration |
| Robert   | Tyler    | Administration |
| Susan    | Hawthorne| Operations     |
| John     | Gooding  | Loans          |
| Helen    | Fleming  | Operations     |
| Chris    | Tucker   | Operations     |
| Sarah    | Parker   | Operations     |
| Jane     | Grossman | Operations     |
| Paula    | Roberts  | Operations     |
| Thomas   | Ziegler  | Operations     |
| Samantha | Jameson  | Operations     |
| John     | Blake    | Operations     |
| Cindy    | Mason    | Operations     |
```

```
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
+----------+-----------+----------------+
18 rows in set (0.01 sec)
```

using 实际上只能在某些特殊情况下起到简化语法的作用，因此本书建议最好始终使用 on 子句以避免不一致的用法可能造成的困扰。

5.1.3　ANSI 连接语法

本书在连接表时使用的语法都符合 SQL92 版本的 ANSI SQL 标准。所有的主流数据库（Oracle Database、Microsoft SQL Server、MySQL、IBM DB2 Universal Database 和 Sybas Adaptive Server）都采用了 SQL92 的连接语法。但由于这些数据库大多都出现在 SQL92 标准发布之前，它们同样也容许一些旧的连接语法。例如，所有这些服务器都能识别并处理下面的查询：

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e, department d
    -> WHERE e.dept_id = d.dept_id;
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
| Susan    | Hawthorne | Operations     |
| John     | Gooding   | Loans          |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
+----------+-----------+----------------+
18 rows in set (0.01 sec)
```

这种旧式的连接方法并不包含 on 子句，而是在 from 子句中定义各表的别名，并使用逗号隔开，然后在 where 子句中包含连接条件。也许你更习惯旧的连接语法而决定不采用 SQL92 的语法，但 ANSI 连接语法具备下列优点：

- 连接条件和过滤条件被分隔到两个子句中（on 子句和 where 子句），使查询语句更易于理解；
- 每两个表之间的连接条件都在它们自己的 on 子句中列出，这样不容易错误地忽略了某些连接条件；
- 使用 SQL92 连接语法的查询语句可以在各种数据库服务器中通用，而旧的语法在不同的服务器上的表现可能略有不同。

SQL92 连接语法的优势在同时包含连接和过滤条件的复杂查询中表现得更明显。考虑下面的查询，它返回 Woburn 支行中所有熟练柜员（在 2007 年以前入职的柜员）开设的账户：

```
mysql> SELECT a.account_id, a.cust_id, a.open_date, a.product_cd
    -> FROM account a, branch b, employee e
    -> WHERE a.open_emp_id = e.emp_id
    ->   AND e.start_date < '2007-01-01'
    ->   AND e.assigned_branch_id = b.branch_id
    ->   AND (e.title = 'Teller' OR e.title = 'Head Teller')
    ->   AND b.name = 'Woburn Branch';
+------------+---------+------------+------------+
| account_id | cust_id | open_date  | product_cd |
+------------+---------+------------+------------+
|          1 |       1 | 2000-01-15 | CHK        |
|          2 |       1 | 2000-01-15 | SAV        |
|          3 |       1 | 2004-06-30 | CD         |
|          4 |       2 | 2001-03-12 | CHK        |
|          5 |       2 | 2001-03-12 | SAV        |
|         17 |       7 | 2004-01-12 | CD         |
|         27 |      11 | 2004-03-22 | BUS        |
+------------+---------+------------+------------+
7 rows in set (0.00 sec)
```

使用这种查询，一是不太容易识别 where 子句中的条件哪些是连接条件，哪些是过滤条件；二是对于使用了何种连接类型也不是显而易见的（如果想确认所使用的连接类型，就需要仔细观察 where 子句中的连接条件中是否包含了特定字符），此外还容易错误地遗漏某些连接条件。下面是返回同样结果的查询，但使用了 SQL92 连接语法：

```
mysql> SELECT a.account_id, a.cust_id, a.open_date, a.product_cd
    -> FROM account a INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    ->   INNER JOIN branch b
    ->   ON e.assigned_branch_id = b.branch_id
    -> WHERE e.start_date < '2007-01-01'
    ->   AND (e.title = 'Teller' OR e.title = 'Head Teller')
    ->   AND b.name = 'Woburn Branch';
+------------+---------+------------+------------+
| account_id | cust_id | open_date  | product_cd |
```

```
+------------+---------+-----------+------------+
|         1  |     1   | 2000-01-15 | CHK        |
|         2  |     1   | 2000-01-15 | SAV        |
|         3  |     1   | 2004-06-30 | CD         |
|         4  |     2   | 2001-03-12 | CHK        |
|         5  |     2   | 2001-03-12 | SAV        |
|        17  |     7   | 2004-01-12 | CD         |
|        27  |     11  | 2004-03-22 | BUS        |
+------------+---------+-----------+------------+
7 rows in set (0.05 sec)
```

如上所示，显然这种使用 SQL92 连接语法的查询方式更易于理解。

5.2　连接 3 个或更多的表

连接 3 个表的方法与连接两个表类似，只是有些许不同。在两个表的连接查询中，from 子句包含了两个表名和一种连接类型，以及一个 on 子句以指定两表是如何连接的。对于 3 个表的连接，在 from 子句中将包含 3 个表和两种连接类型，以及两个 on 子句。下面首先给出另一个两表连接查询的例子：

```
mysql> SELECT a.account_id, c.fed_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    -> WHERE c.cust_type_cd = 'B';
+------------+-----------+
| account_id | fed_id    |
+------------+-----------+
|       24   | 04-1111111 |
|       25   | 04-1111111 |
|       27   | 04-2222222 |
|       28   | 04-3333333 |
|       29   | 04-4444444 |
+------------+-----------+
5 rows in set (0.15 sec)
```

该查询返回所有商务账户（账户类型为‘B’）的账户 ID 和税务号码，看起来相当简洁。如果需要再增加 employee 表以查询开设此账户的柜员姓名，就要采用下面的方法：

```
mysql> SELECT a.account_id, c.fed_id, e.fname, e.lname
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    ->   INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    -> WHERE c.cust_type_cd = 'B';
+------------+-----------+---------+---------+
| account_id | fed_id    | fname   | lname   |
+------------+-----------+---------+---------+
|       24   | 04-1111111 | Theresa | Markham |
```

```
|        25   | 04-1111111 | Theresa | Markham |
|        27   | 04-2222222 | Paula   | Roberts |
|        28   | 04-3333333 | Theresa | Markham |
|        29   | 04-4444444 | John    | Blake   |
+------------+------------+---------+---------+
5 rows in set (0.00 sec)
```

现在 from 子句中包含了 3 个表、两种连接类型和两个 on 子句，因此查询语句看起来有些复杂。由于 from 子句中 3 个表出现的次序是 account 表、customer 表和 employee 表，因此乍看上去可能会认为 employee 表与 customer 表相连接，但是你交换前两个表出现的次序，仍会获得完全一样的查询结果：

```
mysql> SELECT a.account_id, c.fed_id, e.fname, e.lname
    -> FROM customer c INNER JOIN account a
    ->   ON a.cust_id = c.cust_id
    ->   INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+---------+---------+
| account_id | fed_id     | fname   | lname   |
+------------+------------+---------+---------+
|        24  | 04-1111111 | Theresa | Markham |
|        25  | 04-1111111 | Theresa | Markham |
|        27  | 04-2222222 | Paula   | Roberts |
|        28  | 04-3333333 | Theresa | Markham |
|        29  | 04-4444444 | John    | Blake   |
+------------+------------+---------+---------+
5 rows in set (0.09 sec)
```

现在 customer 表被列在首位，然后是 account 表和 employee 表，但 on 子句并没有发生变化，因此查询结果是相同的。让我们把实验进行到底，下面的查询将表的次序完全颠倒过来（从 employee 表到 account 表再到 customer 表）：

```
mysql> SELECT a.account_id, c.fed_id, e.fname, e.lname
    -> FROM employee e INNER JOIN account a
    ->   ON e.emp_id = a.open_emp_id
    ->   INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+---------+---------+
| account_id | fed_id     | fname   | lname   |
+------------+------------+---------+---------+
|        24  | 04-1111111 | Theresa | Markham |
|        25  | 04-1111111 | Theresa | Markham |
|        27  | 04-2222222 | Paula   | Roberts |
|        28  | 04-3333333 | Theresa | Markham |
|        29  | 04-4444444 | John    | Blake   |
+------------+------------+---------+---------+
5 rows in set (0.00 sec)
```

连接的顺序重要吗？

如果你对这 3 个版本的三表查询都产生同样的结果感到疑惑，那么需要记住 SQL 是一种非过程化的语言，也就是说只需要描述要获取的数据库对象，而如何以最好的方式执行查询则由数据库服务器负责。服务器根据所收集的数据库对象信息，在 3 个表中选择一个作为开始点（所选择的表被称为驱动表），然后确定连接其他表的顺序。因此，在 from 子句中各表出现的顺序并不重要。

不过，如果你希望在查询中以特定的顺序连接各表，那么需要将表按照所需要的顺序排列好，然后指定 MySQL 中的 STRAIGHT_JOIN 关键字，或者 SQL Server 中的 FORCE ORDER 选项，或者在 Oracle 数据库中使用 ORDERED 或 LEADING 优化器提示。举个例子，如果要求 MySQL 服务器使用 customer 表作为驱动表，然后连接 account 和 employee 表，那么可以采取下面的做法:

```
mysql> SELECT STRAIGHT_JOIN a.account_id, c.fed_id, e.fname, e.lname
    -> FROM customer c INNER JOIN account a
    ->   ON a.cust_id = c.cust_id
    ->   INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    -> WHERE c.cust_type_cd = 'B';
```

可以将 3 个或更多表的连接视为“滚雪球”，前两个表形成一个开始滚动的“雪球”，而之后的每个表都在“雪球”滚动时依附在上面。也可以将中间结果集看做“雪球”，因为它随着表不断被连接，包含了越来越多的列。因此，实际上 employee 表并没有与 account 表连接，而是与 customer 表和 account 表连接后产生的中间结果集连接。（也许你会诧异我为什么会在这里选择用雪球来比喻，因为在写作本章时我正生活在 New England 的深冬：外面的积雪已经有 110 英寸，明天还会继续下雪。噢，真是令人喜悦啊。）

5.2.1 将子查询结果作为查询表

前面已经介绍了几个使用 3 个表的例子，但这里还有一点值得探讨的内容：如何处理一部分数据集是由子查询所产生的情况。在第 9 章中将聚集于子查询相关的主题，但在前面的章节中已经介绍了一些在 from 子句中使用子查询的概念。下面是前面查询（查找所有 Woburn 支行中有经验的柜员所开设的账户）的另一个版本，其中 account 表与子查询的结果相连接，而不是与 branch 表及 employee 表连接：

```
1 SELECT a.account_id, a.cust_id, a.open_date, a.product_cd
2 FROM account a INNER JOIN
3  (SELECT emp_id, assigned_branch_id
4    FROM employee
5    WHERE start_date < '2007-01-01'
6      AND (title = 'Teller' OR title = 'Head Teller')) e
```

```
7   ON a.open_emp_id = e.emp_id
8   INNER JOIN
9    (SELECT branch_id
10     FROM branch
11     WHERE name = 'Woburn Branch') b
12   ON e.assigned_branch_id = b.branch_id;
```

从第 3 行开始的第一个子查询的别名为 e，它用于查找所有有经验的柜员。从第 9 行开始的第二个子查询的别名为 b，它用于查找 Woburn 支行的 ID。首先，account 表与使用 employee ID 与子查询 e 相连接，然后使用 branch ID 与子查询 b 相连接。该查询的结果与前一个版本（请试着往前找一下）完全相同，但它们的语句看起来却完全不同。

这并不是什么令人惊讶的事，但可能还是需要花上一分钟来搞清楚到底发生了什么。注意，主查询中缺少了 where 子句，因为所有的过滤条件都是针对 employee 表和 branch 表，并且都包含于子查询中，所以主查询中并不需要任何过滤条件。下面单独运行子查询，看一下中间结果集，以使过程更加清楚。下面为第一个对 employee 表的子查询的结果：

```
mysql> SELECT emp_id, assigned_branch_id
    -> FROM employee
    -> WHERE start_date < '2007-01-01'
    ->   AND (title = 'Teller' OR title = 'Head Teller');
+--------+--------------------+
| emp_id | assigned_branch_id |
+--------+--------------------+
|      8 |                  1 |
|      9 |                  1 |
|     10 |                  2 |
|     11 |                  2 |
|     13 |                  3 |
|     14 |                  3 |
|     16 |                  4 |
|     17 |                  4 |
|     18 |                  4 |
+--------+--------------------+
9 rows in set (0.03 sec)
```

如上所示，结果集包含了 employee ID 集合以及相应的 branch ID。当它们通过 emp_id 列与 account 表连接后，所产生的中间结果集包含了所有 account 表的行，并且每行附加了一列，以存放开设该账户的柜员所在支行的 ID。下面是第二个针对 branch 表的子查询结果：

```
mysql> SELECT branch_id
    -> FROM branch
    -> WHERE name = 'Woburn Branch';
+-----------+
| branch_id |
```

```
+-----------+
|       2   |
+-----------+
1 row in set (0.02 sec)
```

该查询返回了只有 1 列的 1 行数据：Woburn 支行的 ID。该表通过 assigned_branch_id 列与中间结果集连接，并在最终结果集中过滤掉所有非 Woburn 支行的柜员所开设的账户。

5.2.2 连续两次使用同一个表

在连接多个表时，有时候可能需要多次连接同一个表。例如，在示例数据库中，branch 表可能存放着 account 表（表示账户的开户支行）和 employee 表的外键（表示雇员的所在支行）。如果需要在结果集中包含这两个支行名称，就需要在 from 子句中两次引用 branch 表，一次与 employee 表连接，另一次与 account 表连接。为了使之正常工作，需要给每个 branch 表的实例定义不同的别名，以便服务器能够在各子句中正确地引用它们，如下所示。

```
mysql> SELECT a.account_id, e.emp_id,
    ->  b_a.name open_branch, b_e.name emp_branch
    -> FROM account a INNER JOIN branch b_a
    ->  ON a.open_branch_id = b_a.branch_id
    ->  INNER JOIN employee e
    ->  ON a.open_emp_id = e.emp_id
    ->  INNER JOIN branch b_e
    ->  ON e.assigned_branch_id = b_e.branch_id
    -> WHERE a.product_cd = 'CHK';
+------------+--------+--------------+--------------+
| account_id | emp_id | open_branch  | emp_branch   |
+------------+--------+--------------+--------------+
|       10   |     1  | Headquarters | Headquarters |
|       14   |     1  | Headquarters | Headquarters |
|       21   |     1  | Headquarters | Headquarters |
|        1   |    10  | Woburn Branch | Woburn Branch |
|        4   |    10  | Woburn Branch | Woburn Branch |
|        7   |    13  | Quincy Branch | Quincy Branch |
|       13   |    16  | So. NH Branch | So. NH Branch |
|       18   |    16  | So. NH Branch | So. NH Branch |
|       24   |    16  | So. NH Branch | So. NH Branch |
|       28   |    16  | So. NH Branch | So. NH Branch |
+------------+--------+--------------+--------------+
10 rows in set (0.16 sec)
```

该查询显示了每个 checking 账户的开户柜员、账户的开户支行以及开户柜员当前所在支行。其中，branch 表被包含了两次，别名分别为 b_a 和 b_e。通过为 branch 表的每个实例指定不同的别名，服务器能够区分所引用的实例：一个用于连接 account 表，另一

个用于连接 employee 表。因此，这是查询中需要使用表别名的又一个例子。

5.3 自连接

不仅可以在同一查询中多次包含同一个表，还可以对表自身进行连接。乍看起来可能会觉得有些奇怪，但有时还是存在这样做的理由的。举例来说，employee 表包含了一个指向自身的外键，即指向本表主键的列（superior_emp_id）。该列指向了雇员的主管（除非该雇员属于领导层，这种情况下该列应为 null）。使用自连接，可以在列出每个雇员姓名的同时列出主管的姓名：

```
mysql> SELECT e.fname, e.lname,
    ->   e_mgr.fname mgr_fname, e_mgr.lname mgr_lname
    -> FROM employee e INNER JOIN employee e_mgr
    ->   ON e.superior_emp_id = e_mgr.emp_id;
+----------+-----------+-----------+-----------+
| fname    | lname     | mgr_fname | mgr_lname |
+----------+-----------+-----------+-----------+
| Susan    | Barker    | Michael   | Smith     |
| Robert   | Tyler     | Michael   | Smith     |
| Susan    | Hawthorne | Robert    | Tyler     |
| John     | Gooding   | Susan     | Hawthorne |
| Helen    | Fleming   | Susan     | Hawthorne |
| Chris    | Tucker    | Helen     | Fleming   |
| Sarah    | Parker    | Helen     | Fleming   |
| Jane     | Grossman  | Helen     | Fleming   |
| Paula    | Roberts   | Susan     | Hawthorne |
| Thomas   | Ziegler   | Paula     | Roberts   |
| Samantha | Jameson   | Paula     | Roberts   |
| John     | Blake     | Susan     | Hawthorne |
| Cindy    | Mason     | John      | Blake     |
| Frank    | Portman   | John      | Blake     |
| Theresa  | Markham   | Susan     | Hawthorne |
| Beth     | Fowler    | Theresa   | Markham   |
| Rick     | Tulman    | Theresa   | Markham   |
+----------+-----------+-----------+-----------+
17 rows in set (0.00 sec)
```

该查询包含两个 employee 表的实例：一个用于提供雇员姓名（表别名为 e），另一个提供主管姓名（表别名为 e_mgr）。on 子句中使用这些别名并借助于 superior_emp_id 外键实现 employee 表的自连接。这是又一个在查询中需要定义表别名的例子，否则，服务器将无法确定在连接条件中所指代的是雇员还是雇员的主管。

employee 表中一共有 18 行，但此查询只返回了 17 行，这是由于银行的总经理 Michael Smith 并没有自己的主管（他的 superior_emp_id 列为 null），因此在该行上的连接失败了。为了在结果集中包含 Michael Smith，必须使用外连接，在第 10 章中对此内容进行

了详述。

5.4 相等连接和不等连接

本章到目前为止的多表查询使用的都是相等连接，即两个表中匹配项的值必须相等。相等连接总是使用等号，例如：

```
ON e.assigned_branch_id = b.branch_id
```

大多数查询使用的是相等连接，但有时也可以通过限定值的范围实现对表的连接，亦即不等连接。下例中的查询使用范围值进行连接：

```
SELECT e.emp_id, e.fname, e.lname, e.start_date
FROM employee e INNER JOIN product p
  ON e.start_date >= p.date_offered
  AND e.start_date <= p.date_retired
WHERE p.name = 'no-fee checking';
```

该查询连接的两个表并没有外键关联,其意图是找出所有在 no-fee checking 产品存续期间入职的银行雇员。因此，雇员的入职时间必须位于产品的提供日期与产品的结束日期之间。

有时候还可能需要不等的自连接，即对表自身使用不等连接。举例来说，假如执行经理决定举办一次面向银行柜员的象棋锦标赛，你被要求创建所有对弈者的列表，这就需要为所有的柜员（title = 'Teller'）进行 employee 表的自连接，并返回所有 emp_id 不同的行（因为柜员无法和自己下棋）：

```
mysql> SELECT e1.fname, e1.lname, 'VS' vs, e2.fname, e2.lname
    -> FROM employee e1 INNER JOIN employee e2
    ->   ON e1.emp_id != e2.emp_id
    -> WHERE e1.title = 'Teller' AND e2.title = 'Teller';
+----------+----------+----+----------+----------+
| fname    | lname    | vs | fname    | lname    |
+----------+----------+----+----------+----------+
|Sarah     | Parker   | VS | Chris    | Tucker   |
|Jane      | Grossman | VS | Chris    | Tucker   |
|Thomas    | Ziegler  | VS | Chris    | Tucker   |
|Samantha  | Jameson  | VS | Chris    | Tucker   |
|Cindy     | Mason    | VS | Chris    | Tucker   |
|Frank     | Portman  | VS | Chris    | Tucker   |
|Beth      | Fowler   | VS | Chris    | Tucker   |
|Rick      | Tulman   | VS | Chris    | Tucker   |
|Chris     | Tucker   | VS | Sarah    | Parker   |
|Jane      | Grossman | VS | Sarah    | Parker   |
|Thomas    | Ziegler  | VS | Sarah    | Parker   |
|Samantha  | Jameson  | VS | Sarah    | Parker   |
|Cindy     | Mason    | VS | Sarah    | Parker   |
|Frank     | Portman  | VS | Sarah    | Parker   |
```

```
|Beth       | Fowler   | VS | Sarah     | Parker    |
|Rick       | Tulman   | VS | Sarah     | Parker    |
...
|Chris      | Tucker   | VS | Rick      | Tulman    |
|Sarah      | Parker   | VS | Rick      | Tulman    |
|Jane       | Grossman | VS | Rick      | Tulman    |
|Thomas     | Ziegler  | VS | Rick      | Tulman    |
|Samantha   | Jameson  | VS | Rick      | Tulman    |
|Cindy      | Mason    | VS | Rick      | Tulman    |
|Frank      | Portman  | VS | Rick      | Tulman    |
|Beth       | Fowler   | VS | Rick      | Tulman    |
+----------+----------+----+----------+----------+
72 rows in set (0.01 sec)
```

现在离正确的目标已经很近了，但这里还存在一个错误，即对于每对比赛选手（比如Sarah Parker 对 Chris Tucker）都有一个与之相反的选手对（如 Chris Tucker 对 Sarah Parker）。可以使用连接条件 e1.emp_id < e2.emp_id 来得到所期望的结果，即每个柜员只会被安排与 employee ID 比他高的柜员比赛（如果你愿意，也可以使用 e1.emp_id > e2.emp_id）：

```
mysql> SELECT e1.fname, e1.lname, 'VS' vs, e2.fname, e2.lname
   -> FROM employee e1 INNER JOIN employee e2
   ->   ON e1.emp_id < e2.emp_id
   -> WHERE e1.title = 'Teller' AND e2.title = 'Teller';
+----------+----------+----+---------+----------+
| fname    | lname    | vs | fname   | lname    |
+----------+----------+----+---------+----------+
| Chris    | Tucker   | VS | Sarah   | Parker   |
| Chris    | Tucker   | VS | Jane    | Grossman |
| Sarah    | Parker   | VS | Jane    | Grossman |
| Chris    | Tucker   | VS | Thomas  | Ziegler  |
| Sarah    | Parker   | VS | Thomas  | Ziegler  |
| Jane     | Grossman | VS | Thomas  | Ziegler  |
| Chris    | Tucker   | VS | Samantha| Jameson  |
| Sarah    | Parker   | VS | Samantha| Jameson  |
| Jane     | Grossman | VS | Samantha| Jameson  |
| Thomas   | Ziegler  | VS | Samantha| Jameson  |
| Chris    | Tucker   | VS | Cindy   | Mason    |
| Sarah    | Parker   | VS | Cindy   | Mason    |
| Jane     | Grossman | VS | Cindy   | Mason    |
| Thomas   | Ziegler  | VS | Cindy   | Mason    |
| Samantha | Jameson  | VS | Cindy   | Mason    |
| Chris    | Tucker   | VS | Frank   | Portman  |
| Sarah    | Parker   | VS | Frank   | Portman  |
| Jane     | Grossman | VS | Frank   | Portman  |
| Thomas   | Ziegler  | VS | Frank   | Portman  |
| Samantha | Jameson  | VS | Frank   | Portman  |
| Cindy    | Mason    | VS | Frank   | Portman  |
| Chris    | Tucker   | VS | Beth    | Fowler   |
```

```
| Sarah    | Parker   | VS | Beth    | Fowler   |
| Jane     | Grossman | VS | Beth    | Fowler   |
| Thomas   | Ziegler  | VS | Beth    | Fowler   |
| Samantha | Jameson  | VS | Beth    | Fowler   |
| Cindy    | Mason    | VS | Beth    | Fowler   |
| Frank    | Portman  | VS | Beth    | Fowler   |
| Chris    | Tucker   | VS | Rick    | Tulman   |
| Sarah    | Parker   | VS | Rick    | Tulman   |
| Jane     | Grossman | VS | Rick    | Tulman   |
| Thomas   | Ziegler  | VS | Rick    | Tulman   |
| Samantha | Jameson  | VS | Rick    | Tulman   |
| Cindy    | Mason    | VS | Rick    | Tulman   |
| Frank    | Portman  | VS | Rick    | Tulman   |
| Beth     | Fowler   | VS | Rick    | Tulman   |
+----------+----------+----+---------+----------+
36 rows in set (0.00 sec)
```

现在列表中包含了 36 个选手对，该数字与从 9 个独立物品（一共 9 个柜员）中选择 2 个的方法数相同。

5.5 连接条件和过滤条件

现在你对在 on 子句中使用连接条件,以及在 where 子句中使用过滤条件已经很熟悉了。SQL 对于放置条件的位置是很灵活的，因此需要仔细构建所需要的查询。举例来说，下面的查询使用一个用于连接两个表的连接条件以及在一个 where 子句中的过滤条件：

```
mysql> SELECT a.account_id, a.product_cd, c.fed_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+------------+
| account_id | product_cd | fed_id     |
+------------+------------+------------+
|         24 | CHK        | 04-1111111 |
|         25 | BUS        | 04-1111111 |
|         27 | BUS        | 04-2222222 |
|         28 | CHK        | 04-3333333 |
|         29 | SBL        | 04-4444444 |
+------------+------------+------------+
5 rows in set (0.01 sec)
```

这看起来相当简单，但是如果错误地将过滤条件放到 on 子句中而不是 where 子句中会怎样呢?

```
mysql> SELECT a.account_id, a.product_cd, c.fed_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    ->     AND c.cust_type_cd = 'B';
```

```
+------------+------------+------------+
| account_id | product_cd | fed_id     |
+------------+------------+------------+
|         24 | CHK        | 04-1111111 |
|         25 | BUS        | 04-1111111 |
|         27 | BUS        | 04-2222222 |
|         28 | CHK        | 04-3333333 |
|         29 | SBL        | 04-4444444 |
+------------+------------+------------+
5 rows in set (0.01 sec)
```

如上所示，第二个版本的查询在 on 子句中包含了两个条件并且省略了 where 子句，但产生的查询结果是相同的。如果两个条件都放到 where 子句但 from 子句仍然使用 ANSI 连接语法会怎样呢?

```
mysql> SELECT a.account_id, a.product_cd, c.fed_id
    -> FROM account a INNER JOIN customer c
    -> WHERE a.cust_id = c.cust_id
    ->   AND c.cust_type_cd = 'B';
+------------+------------+------------+
| account_id | product_cd | fed_id     |
+------------+------------+------------+
|         24 | CHK        | 04-1111111 |
|         25 | BUS        | 04-1111111 |
|         27 | BUS        | 04-2222222 |
|         28 | CHK        | 04-3333333 |
|         29 | SBL        | 04-4444444 |
+------------+------------+------------+
5 rows in set (0.01 sec)
```

MySQL 服务器再一次产生了相同的结果集。尽管如此，你最好还是在合适的位置放置查询条件，以使查询语句易于理解和维护。

5.6 小测验

下面的练习用于测试你对内连接的理解程度。附录 C 提供了这些练习的标准答案。

练习 5-1

给下面的查询填空（使用<#>标记），以获得其后的结果。

```
mysql> SELECT e.emp_id, e.fname, e.lname, b.name
   -> FROM employee e INNER JOIN <1> b
   ->   ON e.assigned_branch_id = b.<2>;
+--------+----------+----------+--------------+
| emp_id | fname    | lname    | name         |
+--------+----------+----------+--------------+
|     1  | Michael  | Smith    | Headquarters |
```

```
|      2  | Susan    | Barker    | Headquarters  |
|      3  | Robert   | Tyler     | Headquarters  |
|      4  | Susan    | Hawthorne | Headquarters  |
|      5  | John     | Gooding   | Headquarters  |
|      6  | Helen    | Fleming   | Headquarters  |
|      7  | Chris    | Tucker    | Headquarters  |
|      8  | Sarah    | Parker    | Headquarters  |
|      9  | Jane     | Grossman  | Headquarters  |
|     10  | Paula    | Roberts   | Woburn Branch |
|     11  | Thomas   | Ziegler   | Woburn Branch |
|     12  | Samantha | Jameson   | Woburn Branch |
|     13  | John     | Blake     | Quincy Branch |
|     14  | Cindy    | Mason     | Quincy Branch |
|     15  | Frank    | Portman   | Quincy Branch |
|     16  | Theresa  | Markham   | So. NH Branch |
|     17  | Beth     | Fowler    | So. NH Branch |
|     18  | Rick     | Tulman    | So. NH Branch |
+--------+----------+-----------+---------------+
18 rows in set (0.03 sec)
```

练习 5-2

编写查询，返回所有非商务顾客的账户 ID（customer.cust_type_cd = 'I'）、顾客的联邦个人识别号码（cusomer.fed_id）以及账户所依赖的产品名称（product.name）。

练习 5-3

构建查询，查找所有主管位于另一个部门的雇员，需要获取该雇员的 ID、姓氏和名字。

第 6 章

使用集合

尽管可以在与数据库交互时一次只处理一行数据，但实际上关系数据库通常处理的都是数据集合。前面已经介绍了如何通过查询及子查询创建表、通过 insert 语句持久化数据以及使用 join 将它们连接到一起，本章将展示如何使用各种集合操作符来联合多个表。

6.1　集合理论基础

世界上许多地方都将基础集合理论作为入门级数学教程，图 6-1 或许能唤回你对集合知识的一些记忆。

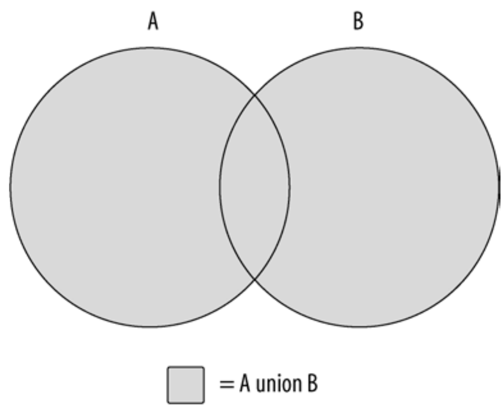


图 6-1　集合的并操作

图 6-1 中的阴影部分代表了集合 A 和集合 B 的联合，即两个集合的并集（其中重复元素只包含一次）。该图你看起来眼熟吗？如果是，那么你曾学过的知识将要派上用场了；

如果不是也无需担心，因为该图表达的含义是十分形象化并易于理解的。

图 6-1 中使用圆圈代表数据集合（A 和 B），可以想象，两个集合所共有的子集合数据在图中由重叠区域所代表。由于集合理论对于两个无数据重合的数据集合不感兴趣，因此下文将继续使用图示来说明集合的相关操作。另一种集合操作只关心两个数据集的重合部分，即集合的交操作（intersect），如图 6-2 所示。

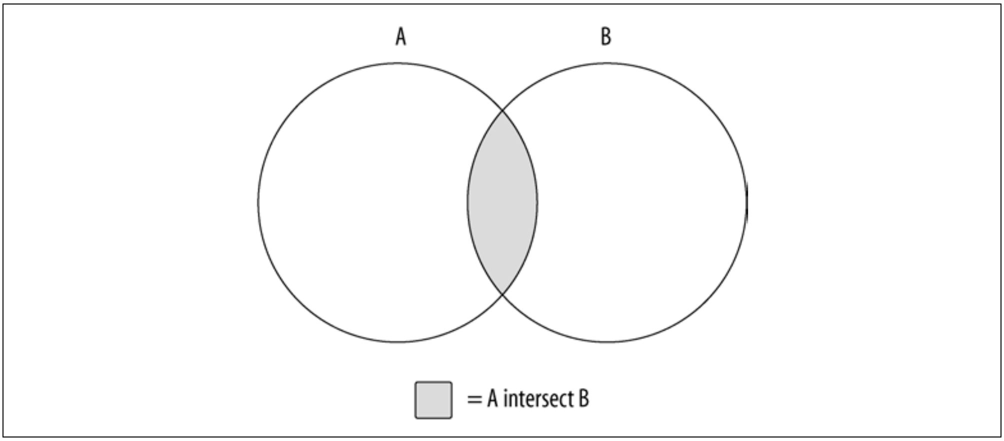


图 6-2　集合的交操作

集合 A 和 B 的交操作产生的数据集合只包括两个集合的重合区域，如果两个集合没有重合的数据，那么交操作将产生空集。

图 6-3 展示了第三个也是最后一个集合操作，即差操作（except）。

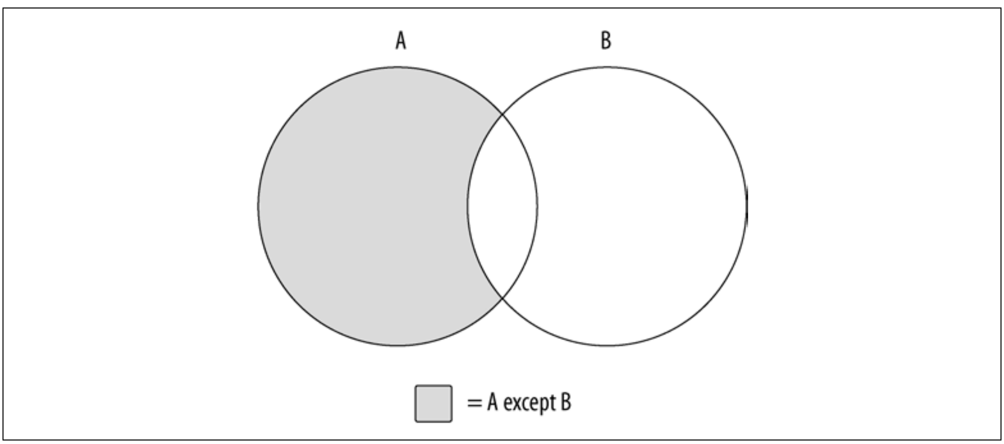


图 6-3　集合的差操作

图 6-3 显示了集合 A 减去集合 B 的结果，即在完整的集合 A 中除去与集合 B 重合的数

据。如果两个集合没有重合部分，那么 A 减 B 的差操作所产生的结果为整个集合 A。

使用这 3 种操作，或者联合其他的集合操作，可以产生任何所需要的结果，比如图 6-4 中所表示的集合。

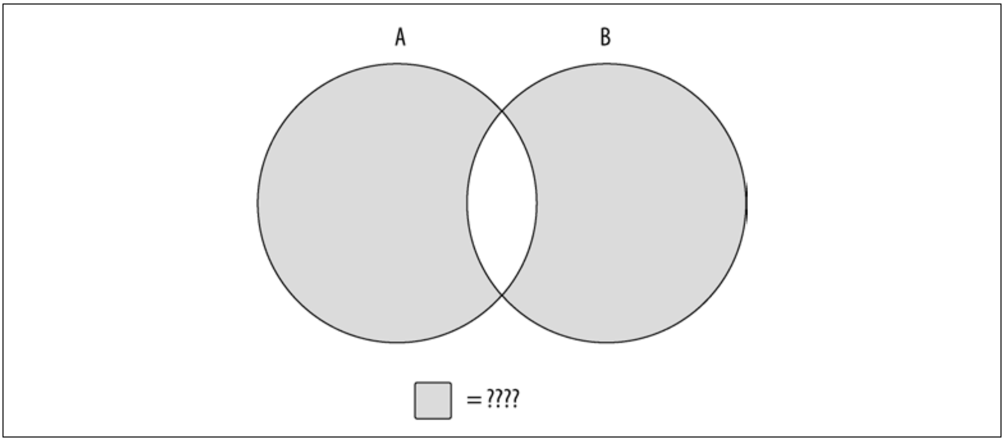


图 6-4　未知的数据集

该数据集包括集合 A 和 B 中所有非重合区域的部分。可以只使用前面介绍的 3 种操作产生这个结果，即先构造一个包含集合 A 和 B 所有元素的数据集，再使用第二个操作去除重复的区域。如果使用 A union B 表示两个集合的并集，A intersect B 表示它们的交集，那么可以使用下面的方法产生图 6-4 中的数据集合：

(A union B) except (A intersect B)

当然，通常获取同一个结果的方法不止一个，下面的操作所产生的结果也是等价的：

(A except B) union (B except A)

使用图示的方法可以使这些概念非常易于理解，后面将说明如何在关系数据库中使用 SQL 集合操作符来应用它们。

6.2　集合理论实践

上节的图示中使用圆圈代表数据集合，但并没有表达数据集中所包含的数据。当处理实际的数据时，还需要了解数据集的结构。举例来说，假设需要产生 product 表和 customer 表的并集，这两个表的定义如下所示。

```
mysql> DESC product;
+-----------------+------------+------+-----+---------+-------+
| Field           | Type       | Null | Key | Default | Extra |
+-----------------+------------+------+-----+---------+-------+
```

```
| product_cd      | varchar(10) | NO   | PRI | NULL    |       |
| name            | varchar(50) | NO   |     | NULL    |       |
| product_type_cd | varchar(10) | NO   | MUL | NULL    |       |
| date_offered    | date        | YES  |     | NULL    |       |
| date_retired    | date        | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+
5 rows in set (0.23 sec)
mysql> DESC customer;
+-----------------+-------------+------+-----+---------+------------+
| Field           | Type        |Null  | Key | Default |   Extra    |
+-----------------+-------------+------+-----+---------+------------+
| cust_id         |int(10)unsigned| NO   | PRI | NULL    |auto_increment|
| fed_id          | varchar(12) | NO   |     |NULL     |            |
| cust_type_cd    | enum('I','B')| NO   |     |NULL     |            |
| address         | varchar(30) | YES  |     |NULL     |            |
| city            | varchar(20) | YES  |     |NULL     |            |
| state           | varchar(20) | YES  |     |NULL     |            |
| postal_code     | varchar(10) | YES  |     |NULL     |            |
+-----------------+-------------+------+-----+---------+------------+
7 rows in set (0.04 sec)
```

当完成对这两个表的连接后，所产生的结果表中的第一列应为 product.product_cd 和 customer.cust_id 列的组合，第二列为 product.name 和 customer.fed_id 的组合，依此类推。其中，一些列对是易于组合的（比如两列都为数字型），另一些列对则难以组合，比如数字列和字符串列或者字符串列和日期列。此外，连接表的第 6 列和第 7 列数据只能来自于 customer 表的第 6 列和第 7 列，因为 product 表只包含 5 列。显然，对于要连接的表必须满足一些共有条件。

因此，当对两个数据集合执行集合操作时，必须首先应用下面的规范。

- 两个数据集合必须具有同样数目的列；
- 两个数据集中对应列的数据类型必须是一样的（或者服务器能够将其中一种转换为另一种）。

在实践中使用这两条规则能够更容易地处理“重合数据”，即对于两张表中的数据行，如果每个列对都包含了同样的字符串、数字或日期等，那么它们可以被视为重复行。

在两条 select 语句中可以使用集合操作符执行集合操作，例如：

```
mysql> SELECT 1 num, 'abc' str
    -> UNION
    -> SELECT 9 num, 'xyz' str;
+-----+-----+
| num | str |
+-----+-----+
|   1 | abc |
```

```
|   9  | xyz |
+-----+-----+
2 rows in set (0.02 sec)
```

每个独立的查询都会产生一个包含单个行的数据集合，该行由一个数字列和一个字符串列组成。集合操作符（本例中为 union）告知数据库服务器连接两个集合的所有行。因此最终的集合包含两个两列的行。该查询被称为复合查询，它将多个独立的查询组合到了一起。本书后面将会看到更复杂的复合查询，它们可能会使用多个集合操作符来组合两个以上的查询，以获取最终的查询结果。

6.3 集合操作符

SQL 语言包含 3 个集合操作符以执行前面章节中所描述的各种集合操作。此外，每个集合操作符可以有两种修饰符：一个表示包含重复项，另一个表示去除重复项（但不一定是所有的重复项）。下面将分别介绍每种操作符的含义与用法。

6.3.1 union 操作符

union 与 union all 操作符可以连接多个数据集，它们的区别在于 union 对连接后的集合排序并去除重复项，而 union all 保留重复项。使用 union all 得到的最终数据集的行数总是等于所要连接的各集合的行数之和，该操作是最易于执行的集合操作（从服务端的观点来看），因为服务器不需要检查重复的数据。下面的例子演示了如何使用 union all 操作符从两个子类型客户表中产生完整的客户数据集合：

```
mysql> SELECT 'IND' type_cd, cust_id, lname name
    -> FROM individual
    -> UNION ALL
    -> SELECT 'BUS' type_cd, cust_id, name
    -> FROM business;
+---------+---------+----------------------+
| type_cd | cust_id | name                 |
+---------+---------+----------------------+
| IND     |       1 | Hadley               |
| IND     |       2 | Tingley              |
| IND     |       3 | Tucker               |
| IND     |       4 | Hayward              |
| IND     |       5 | Frasier              |
| IND     |       6 | Spencer              |
| IND     |       7 | Young                |
| IND     |       8 | Blake                |
| IND     |       9 | Farley               |
| BUS     |      10 | Chilton Engineering  |
| BUS     |      11 | Northeast Cooling Inc.|
| BUS     |      12 | Superior Auto Body   |
| BUS     |      13 | AAA Insurance Inc.   |
+---------+---------+----------------------+
13 rows in set (0.04 sec)
```

该查询返回了所有 13 个客户，其中 9 行来自于 individual 表，而其他 4 行来自于 business 表。由于 business 表中只使用 1 列保存公司名，而 individual 表包含两个姓名列，分别用于姓氏和名字，因此在本例中，只包含 individual 表的姓氏（lname）列。

需要强调的是 union all 操作符并不删除重复项，下面的查询与前一个例子类似，只不过连接了两次 business 表：

```
mysql> SELECT 'IND' type_cd, cust_id, lname name
    -> FROM individual
    -> UNION ALL
    -> SELECT 'BUS' type_cd, cust_id, name
    -> FROM business
    -> UNION ALL
    -> SELECT 'BUS' type_cd, cust_id, name
    -> FROM business;
+---------+---------+----------------------+
| type_cd | cust_id | name                 |
+---------+---------+----------------------+
| IND     |       1 | Hadley               |
| IND     |       2 | Tingley              |
| IND     |       3 | Tucker               |
| IND     |       4 | Hayward              |
| IND     |       5 | Frasier              |
| IND     |       6 | Spencer              |
| IND     |       7 | Young                |
| IND     |       8 | Blake                |
| IND     |       9 | Farley               |
| BUS     |      10 | Chilton Engineering  |
| BUS     |      11 | Northeast Cooling Inc.|
| BUS     |      12 | Superior Auto Body   |
| BUS     |      13 | AAA Insurance Inc.   |
| BUS     |      10 | Chilton Engineering  |
| BUS     |      11 | Northeast Cooling Inc.|
| BUS     |      12 | Superior Auto Body   |
| BUS     |      13 | AAA Insurance Inc.   |
+---------+---------+----------------------+
17 rows in set (0.01 sec)
```

这个复合查询包含了 3 个 select 语句，其中两个是完全一样的。从结果表中可以看出，来自于 business 表的 4 行数据（客户 ID 为 10、11、12 和 13）被包含了两次。

当然在实际的复合查询中包含两个重复的查询并不常见，下面给出另一个返回重复数据的复合查询例子：

```
mysql> SELECT emp_id
    -> FROM employee
```

```
    -> WHERE assigned_branch_id = 2
    ->   AND (title = 'Teller' OR title = 'Head Teller')
    -> UNION ALL
    -> SELECT DISTINCT open_emp_id
    -> FROM account
    -> WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|    10  |
|    11  |
|    12  |
|    10  |
+--------+
4 rows in set (0.01 sec)
```

复合语句中的第一个查询获取分配到 Woburn 支行的所有柜员，而第二个查询返回所有在 Woburn 支行开户的雇员（使用 distinct 去除重复项）。结果集中的 4 行数据中有 1 行是重复的（ID 为 10 的雇员），如果希望连接后的表排除重复行，那么需要使用 union 操作符来替代 union all：

```
mysql> SELECT emp_id
    -> FROM employee
    -> WHERE assigned_branch_id = 2
    ->   AND (title = 'Teller' OR title = 'Head Teller')
    -> UNION
    -> SELECT DISTINCT open_emp_id
    -> FROM account
    -> WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|    10  |
|    11  |
|    12  |
+--------+
3 rows in set (0.01 sec)
```

此查询只在结果集中包含了 3 个独立的行，而不是使用 union all 时所返回的 4 行（3 个独立的，1 个重复的）。

6.3.2 intersect 操作符

ANSI 的 SQL 规范中定义了 intersect 操作符来执行集合交操作，不幸的是，MySQL 6.0 版还未实现 intersect 操作符，不过在 Oracle 或 SQL Server 2008 中可以使用它。由于本书使用 MySQL 作为所有范例，因此本节中的示例查询语句是虚构的，并不能在 MySQL 的任何版本（包括 6.0）中执行。因为这些语句并没有在 MySQL 服务器上实际执行，因此省去了 MySQL 的提示符（mysql>）。

如果复合查询中的两个子查询结果并没有交集，那么 intersect 操作将返回空集。考虑下面的查询：

```
SELECT emp_id, fname, lname
FROM employee
INTERSECT
SELECT cust_id, fname, lname
FROM individual;
Empty set (0.04 sec)
```

第一个查询返回每个雇员的 ID 和姓名，而第二个查询返回每个客户的 ID 和姓名。这两个集合完全是不重合的，因此对它们的交集操作产生的结果为空集。

下面考察两个包含重复数据的集合，并对它们应用 intersect 操作符。这里使用与前面用于说明 union 和 union all 区别相同的例子，只不过这一次使用的是 intersect：

```
SELECT emp_id
FROM employee
WHERE assigned_branch_id = 2
  AND (title = 'Teller' OR title = 'Head Teller')
INTERSECT
SELECT DISTINCT open_emp_id
FROM account
WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|    10  |
+--------+
1 row in set (0.01 sec)
```

这两个查询的交集操作产生的结果为 ID 等于 10 的雇员，即两个待连接查询的结果集中唯一重复的值。

intersect 操作符去除了交集区域中所有重复的行，除此之外，ANSI SQL 规范还定义了并不删除重复行的 intersect all 操作符，不过在当前的数据库服务器中只有 IBM 的 DB2 Universal Server 实现了 intersect all 操作符。

6.3.3　except 操作符

ANSI SQL 规范定义了 except 操作符以执行集合差操作。同样不幸的是，MySQL 6.0 版本仍未实现 except 操作符，因此这里采用与上文同样的方式（虚构 MySQL 语句）进行阐述。

> **提示**
>
> 在 Oracle 数据库中，可以使用非 ANSI 标准的 minus 操作符来替代。

except 操作符返回第一个表减去与第二个表重合的元素后剩下的部分。下面的例子与前一小节相同，只不过使用 except 替代了 intersect：

```
SELECT emp_id
FROM employee
WHERE assigned_branch_id = 2
  AND (title = 'Teller' OR title = 'Head Teller')
EXCEPT
SELECT DISTINCT open_emp_id
FROM account
WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|    11  |
|    12  |
+--------+
2 rows in set (0.01 sec)
```

在该查询中，结果集包含第一个查询所产生的 3 行数据减去两个查询结果集中都包含的雇员数据（ID 为 10）。ANSI SQL 规范中还定义了 except all 操作符，并且也只有 IBM DB2 Universal Server 已经实现了它。

Except all 操作符有点难以理解，因此下面提供一个例子说明它是如何处理重复数据的。假设我们拥有下面两个数据集：

集合 A

```
+--------+
| emp_id |
+--------+
|    10  |
|    11  |
|    12  |
|    10  |
|    10  |
+--------+
```

集合 B

```
+--------+
| emp_id |
+--------+
|    10  |
|    10  |
+--------+
```

操作 A except B 产生如下结果：

```
+--------+
| emp_id |
+--------+
|    11  |
|    12  |
+--------+
```

如果将操作修改为 A except all B，那么结果如下所示。

```
+--------+
| emp_id |
+--------+
|    10  |
|    11  |
|    12  |
+--------+
```

由此可见，两个操作的区别在于，except 在集合 A 中除去所有的重复数据，而 except all 则根据重复数据在集合 B 中出现的次数进行删除。

6.4　集合操作规则

下面几个小节简述一下使用复合查询的一些规则。

6.4.1　对复合查询结果排序

如果需要对复合查询的结果进行排序，那么可以在最后一个查询后面增加 order by 子句。当在 order by 子句中指定要排序的列时，需要从复合查询的第一个查询中选择列名。通常情况下，复合查询中两个查询对应列的名字是相同的，但并不是强制的，如下面的语句所示。

```
mysql> SELECT emp_id, assigned_branch_id
    -> FROM employee
    -> WHERE title = 'Teller'
    -> UNION
    -> SELECT open_emp_id, open_branch_id
    -> FROM account
    -> WHERE product_cd = 'SAV'
    -> ORDER BY emp_id;
+--------+--------------------+
| emp_id | assigned_branch_id |
+--------+--------------------+
|     1  |              1     |
|     7  |              1     |
|     8  |              1     |
|     9  |              1     |
|    10  |              2     |
|    11  |              2     |
|    12  |              2     |
|    14  |              3     |
|    15  |              3     |
|    16  |              4     |
|    17  |              4     |
|    18  |              4     |
+--------+--------------------+
12 rows in set (0.04 sec)
```

本例中两个查询指定的列名并不相同，如果在 order by 子句中指定的是来自第二个查询的列名，那么将发生下面的错误。

```
mysql> SELECT emp_id, assigned_branch_id
    -> FROM employee
    -> WHERE title = 'Teller'
    -> UNION
    -> SELECT open_emp_id, open_branch_id
    -> FROM account
    -> WHERE product_cd = 'SAV'
    -> ORDER BY open_emp_id;
ERROR 1054 (42S22): Unknown column 'open_emp_id' in 'order clause'
```

因此本书建议对两个查询的各列定义不同的列别名，以避免此类错误的发生。

6.4.2 集合操作符优先级

如果复合查询包含两个以上使用不同集合操作符的查询，那么需要在复合语句中确定查询执行的次序，以获取想要的结果。考虑下面包含 3 个查询的复合语句：

```
mysql> SELECT cust_id
    -> FROM account
    -> WHERE product_cd IN ('SAV', 'MM')
    -> UNION ALL
    -> SELECT a.cust_id
    -> FROM account a INNER JOIN branch b
    ->   ON a.open_branch_id = b.branch_id
    -> WHERE b.name = 'Woburn Branch'
    -> UNION
    -> SELECT cust_id
    -> FROM account
    -> WHERE avail_balance BETWEEN 500 AND 2500;
+---------+
| cust_id |
+---------+
|      1  |
|      2  |
|      3  |
|      4  |
|      8  |
|      9  |
|      7  |
|     11  |
|      5  |
+---------+
9 rows in set (0.00 sec)
```

该复合查询包含 3 个返回非唯一客户 ID 的查询，第一个和第二个查询使用 union all 操作符连接，而第二个和第三个查询之间使用的是 union 操作符。尽管表面上看起来 union 和 union all 操作符的区别不大，实际上它们还是有差别的。下面是类似的复合查询，只不过这两个操作符出现的次序被调换了。

```
mysql> SELECT cust_id
    -> FROM account
    -> WHERE product_cd IN ('SAV', 'MM')
    -> UNION
    -> SELECT a.cust_id
    -> FROM account a INNER JOIN branch b
    ->   ON a.open_branch_id = b.branch_id
    -> WHERE b.name = 'Woburn Branch'
    -> UNION ALL
    -> SELECT cust_id
    -> FROM account
    -> WHERE avail_balance BETWEEN 500 AND 2500;
+---------+
| cust_id |
+---------+
|      1  |
|      2  |
|      3  |
|      4  |
|      8  |
|      9  |
|      7  |
|     11  |
|      1  |
|      1  |
|      2  |
|      3  |
|      3  |
|      4  |
|      4  |
|      5  |
|      9  |
+---------+
17 rows in set (0.00 sec)
```

如上面的结果所示，显然用不同的集合操作符构建复合查询会产生不同的查询结果。通常，复合查询包含 3 个或 3 个以上的查询语句，它们以自顶向下的顺序被解析和执行，但还需要注意下面两点：

- 根据 ANSI SQL 标准，在调用集合操作时，intersect 操作符比其他操作符具有更高

的优先级；

- 可以用圆括号对多个查询进行封装，以明确指定它们的执行次序。

虽然 MySQL 还没有实现 intersect 操作符，并且也不允许在复合查询中使用括号，但是仍然需要小心地安排复合查询中各查询的次序，以获得期望的结果。如果使用其他的数据库服务器，可以用括号封装所连接的查询，以改变复合查询中默认的自顶向下的处理顺序，如下所示。

```
(SELECT cust_id
 FROM account
 WHERE product_cd IN ('SAV', 'MM')
 UNION ALL
 SELECT a.cust_id
 FROM account a INNER JOIN branch b
   ON a.open_branch_id = b.branch_id
 WHERE b.name = 'Woburn Branch')
INTERSECT
(SELECT cust_id
 FROM account
 WHERE avail_balance BETWEEN 500 AND 2500
 EXCEPT
 SELECT cust_id
 FROM account
 WHERE product_cd = 'CD'
  AND avail_balance < 1000);
```

在该复合查询中，使用 union all 操作符连接第一个和第二个查询，使用 except 操作符连接第三个和第四个查询。最后用 intersect 操作符连接这两个操作的结果，以产生最终的结果集。

6.5 小测验

下面的练习题用于测试读者对集合操作符的理解程度，答案参见附录 C。

练习 6-1

如果集合 A = {L M N O P}，集合 B = {P Q R S T}，那么下面的操作产生的结果集是什么？

- A union B
- A union all B
- A intersect B
- A except B

练习 6-2

编写一个复合查询，以查找所有个人客户以及雇员的姓氏和名字。

练习 6-3

根据 lname 列对练习 6-2 的结果进行排序。

第 7 章

数据生成、转换和操作

如前言中提到的，本书致力于讲述可以应用到多种数据库服务器上的通用 SQL 技巧，而本章主要处理对字符串、数字或临时数据的生成、转换和操作。由于 SQL 语言本身并不包含这些功能相关的命令，各数据库服务器必须使用内建函数处理数据生成、转换和操作，并且尽管 SQL 标准指定了一些函数，但有的数据库厂商并没有遵从这些函数规范。

因此，本章的目的在于介绍使用 SQL 语句的一些常用方式处理数据，并展示 Microsoft SQL Server、Oracle Database 和 MySQL 的一些内建函数。强烈建议读者在阅读本章时，购买一本所使用数据库服务器的参考手册，其中包含了它的所有内建函数。如果你使用了不只一种数据库服务器，那么可以使用涵盖多种服务器的参考手册，如 Kevin Klin 等的“SQL in a Nutshell”和 Jonathan Gennick 的“SQL Pocket Guide”，它们都是由 O’Reilly 出版的。

7.1　使用字符串数据

当使用字符串数据时，可以使用下面的字符数据类型。

CHAR

　　固定长度、不足部分使用空格填充的字符串。MySQL 的 CHAR 类型的长度为 255，Oracle 数据库允许 2000 个字符，而 SQL Server 的 CHAR 类型允许的最大长度为 8000。

varchar

　　变长字符串。MySQL 允许 varchar 列最多可包含 65536 个字符，Oracle 数据（varchar2 类型）允许包含 4000 个字符，而 SQL Server 允许 varchar 类型的最大长度为 8000。

text（MySQL 和 SQL Server）或 CLOB（Character Large Object; Oracle Database）

容纳大长度的变长字符串（通常在上下文中指代文档）。MySQL 具有多种 text 类型（tinytext、text、mediumtext 和 longtext），最多可保存 4GB 大小的文档数据。SQL Server 只具有一个最大长度为 2GB 的 text 类型，而 Oracle 数据库包含的 CLOB 数据类型可以保存 128TB 的文档。SQL Server 2005 建议使用 varchar(max)数据类型来替代 text 类型，后者在未来的版本中可能被删除。

为了演示如何使用这些字符串类型，首先为本章的各示例建立下面的表：

```
CREATE TABLE string_tbl
 (char_fld CHAR(30),
  vchar_fld VARCHAR(30),
  text_fld TEXT
 );
```

下面两小节将展示如何生成和操作字符串数据。

7.1.1 生成字符串

生成字符串数据的最简单方式是使用一对单引号将字符串包含起来，例如：

```
mysql> INSERT INTO string_tbl (char_fld, vchar_fld, text_fld)
    -> VALUES ('This is char data',
    ->  'This is varchar data',
    ->  'This is text data');
Query OK, 1 row affected (0.00 sec)
```

在向表中插入字符串数据时，需要保证其长度不超出字符列的最大长度（用户指定的或该数据类型内建的最大值），否则服务器将会抛出异常。这是所有数据库服务器的默认做法，但可以通过对 MySQL 或 SQL Server 的配置，将其修改为直接截断字符串后插入，并且不抛出任何异常。为了演示 MySQL 如何处理这种情况，下面使用 update 语句向最大长度为 30 的 vchar_fld 列中插入长度为 46 的字符串：

```
mysql> UPDATE string_tbl
    -> SET vchar_fld = 'This is a piece of extremely long varchar data';
ERROR 1406 (22001): Data too long for column 'vchar_fld' at row 1
```

在 MySQL 6.0 中，默认行为是"strict"模式，即在发生问题时抛出异常，而在早先的服务器版本中，默认方式是截断字符串并发出一个警告。如果希望数据库引擎采取后一种方式，可以将之修改为 ANSI 模式，下例演示了如何查看数据库的当前模式，以及如何使用 SET 命令改变模式：

```
mysql> SELECT @@session.sql_mode;
+----------------------------------------------------------------+
| @@session.sql_mode                                             |
+----------------------------------------------------------------+
| STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+----------------------------------------------------------------+
1 row in set (0.00 sec)
```

```
mysql> SET sql_mode='ansi';
Query OK, 0 rows affected (0.08 sec)

mysql> SELECT @@session.sql_mode;
+-----------------------------------------------------------------+
| @@session.sql_mode                                              |
+-----------------------------------------------------------------+
| REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,ANSI     |
+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

如果再次运行前一个 UPDATE 语句，那么将发现该列内容发生了改变，同时也会产生下面的警告：

```
mysql> SHOW WARNINGS;
+---------+------+------------------------------------------------+
| Level   | Code | Message                                        |
+---------+------+------------------------------------------------+
| Warning | 1265 | Data truncated for column 'vchar_fld' at row 1 |
+---------+------+------------------------------------------------+
1 row in set (0.00 sec)
```

提取 vchar_fld 列的数据，将会看到被截断后的字符串：

```
mysql> SELECT vchar_fld
    -> FROM string_tbl;
+--------------------------------+
| vchar_fld                      |
+--------------------------------+
| This is a piece of extremely l |
+--------------------------------+
1 row in set (0.05 sec)
```

如上所示，长度为 46 的字符串中只有前 30 个字符被保存到 vchar_fld 列。在使用 varchar 列时避免字符串截断（或在 Oracal 和 MySQL 的“strict”模式下抛出异常）的最好方法是将列长度的上限设置为足够大，以处理可能存储在列中的最长字符串（由于服务器是在存储字符串时按需分配空间，因此不会因为将 varchar 列的上限值设置得比较大而浪费资源）。

包含单引号

由于在 SQL 中的字符串使用单引号分隔，因此对于本身包含单引号（或撇号）的字符串会产生警告，比如无法将下面的字符串插入服务器，因为其中单词 doesn’t 中所含的撇号被视为字符串的结束标记：

```
UPDATE string_tbl
SET text_fld = 'This string doesn't work';
```

为了使服务器忽略单词 doesn’t 中的撇号，需要在字符串中加上转义符以便服务器能够将它视为普通字符。上面提到的 3 种数据库服务器都支持通过在单引号前再添加一个

单引号作为转义符，例如：

```
mysql> UPDATE string_tbl
    -> SET text_fld = 'This string didn''t work, but it does now';
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

提示

Oracle和MySQL的用户可以选择使用反斜杠作为单引号的转义符，例如：

```
UPDATE string_tbl SET text_fld =
  'This string didn\'t work, but it does now'
```

在屏幕上或报表域中提取该字符串时，可以不使用任何特殊的方式处理内嵌的引号：

```
mysql> SELECT text_fld
    -> FROM string_tbl;
+------------------------------------------+
| text_fld                                 |
+------------------------------------------+
| This string didn't work, but it does now |
+------------------------------------------+
1 row in set (0.00 sec)
```

当然，在提取字符串并添加到另一个程序所读取的文件中时，或许需要为获取的字符串增加转义符。如果使用 MySQL，可以使用内建的函数 quote()，它用单引号将整个字符串包含起来，并为字符串本身的单引号/撇号增加转义符，下面为使用 quote()函数获取字符串的例子：

```
mysql> SELECT quote(text_fld)
    -> FROM string_tbl;
+---------------------------------------------+
| QUOTE(text_fld)                             |
+---------------------------------------------+
| 'This string didn\'t work, but it does now' |
+---------------------------------------------+
1 row in set (0.04 sec)
```

当提取用于导出的数据时，可以使用 quote()函数处理所有非系统生成的字符列，如 customer_notes 列。

包含特殊字符

如果应用是国际化的，或许需要处理由键盘字符之外的字符所构成的字符串。例如，使用法语或德语时，需要包含重音符号，如 é 和 ö。SQL Server 和 MySQL 服务器包含了内建函数 char()，可用于从 ASCII 字符集中 255 个字符中任意构建字符串（Oracle 数据库用户可以使用 chr()函数）。下面的例子获取一个输入的字符串以及通过独立字符构建的与之相同的字符串：

```
mysql> SELECT 'abcdefg', CHAR(97,98,99,100,101,102,103);
+---------+--------------------------------+
| abcdefg | CHAR(97,98,99,100,101,102,103) |
+---------+--------------------------------+
| abcdefg | abcdefg                        |
+---------+--------------------------------+
1 row in set (0.01 sec)
```

从本例可见，ASCII 字符集中的第 97 个字符为字母 a。上例中的字符并不特殊，下面的例子显示了重音字符以及其他一些特殊字符：

```
mysql> SELECT CHAR(128,129,130,131,132,133,134,135,136,137);
+-----------------------------------------------+
| CHAR(128,129,130,131,132,133,134,135,136,137) |
+-----------------------------------------------+
| Çüéâäàåçêë                                    |
+-----------------------------------------------+
1 row in set (0.01 sec)

mysql> SELECT CHAR(138,139,140,141,142,143,144,145,146,147);
+-----------------------------------------------+
| CHAR(138,139,140,141,142,143,144,145,146,147) |
+-----------------------------------------------+
| èïîìÄÅÉæÆô                                    |
+-----------------------------------------------+
1 row in set (0.01 sec)

mysql> SELECT CHAR(148,149,150,151,152,153,154,155,156,157);
+-----------------------------------------------+
| CHAR(148,149,150,151,152,153,154,155,156,157) |
+-----------------------------------------------+
| öòÛùÿ...Ü¢£¥                                  |
+-----------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CHAR(158,159,160,161,162,163,164,165);
+---------------------------------------+
| CHAR(158,159,160,161,162,163,164,165) |
+---------------------------------------+
| fáíóúñÑ                               |
+---------------------------------------+
1 row in set (0.01 sec)
```

提示

本节中的例子使用的是 latin1 字符集。如果将会话设置为其他字符集，那么上面将显示不同的字符。虽然应用的概念是相同的，但是仍然需要熟悉所用的字符集以定位特殊字符。

根据一个个字符构建字符串是相当烦琐的，特别是其中一些符号是重音符。幸运的是，可以使用 concat()函数来连接若干字符串。其中，一些可以输入，另一些可以通过 char()函数生成。下面举例说明如何使用 concat()和 char()函数构建短语 danke schön：

```
mysql> SELECT CONCAT('danke sch', CHAR(148), 'n');
+-----------------------------------+
| CONCAT('danke sch', CHAR(148), 'n') |
+-----------------------------------+
| danke schön                       |
```

```
+-----------------------------------+
1 row in set (0.00 sec)
```

提示

Oracle 数据库用户可以使用连接操作符（||）来取代 concat()函数，例如：

```
SELECT 'danke sch' || CHR(148) || 'n'
FROM dual;
```

SQL Server 也不提供 concat()函数，可以使用连接字符（+），例如：

```
SELECT 'danke sch' + CHAR(148) + 'n'
```

如果需要根据字符来查找对应的 ASCII 码，可以使用 ascii()函数，它接受一个字符并返回其序号：

```
mysql> SELECT ASCII('ö');
+------------+
| ASCII('ö') |
+------------+
|      148   |
+------------+
1 row in set (0.00 sec)
```

通过 char()、ascii()和 concat()函数（或连接操作符），可以使用任何罗马字符，即使所用的键盘并不包括重音符或其他特殊字符。

7.1.2　操作字符串

每种数据库服务器都包含许多用于操作字符串的内建函数。本小节将主要讨论其中两类字符串函数：返回数字的和返回字符串的。首先需要重设 string_tbl 表中的数据：

```
mysql> DELETE FROM string_tbl;
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO string_tbl (char_fld, vchar_fld, text_fld)
    -> VALUES ('This string is 28 characters',
    ->  'This string is 28 characters',
    ->  'This string is 28 characters');
Query OK, 1 row affected (0.00 sec)
```

返回数字的字符串函数

显然，最常用的返回数字的字符串函数为 length()函数，它返回字符串的字符数（SQL Server 用户需要使用 len()函数）。下面的查询对 string_tbl 表中的每个列都应用一次 length()函数：

```
mysql> SELECT LENGTH(char_fld) char_length,
    ->  LENGTH(vchar_fld) varchar_length,
    ->  LENGTH(text_fld) text_length
    -> FROM string_tbl;
+-------------+----------------+-------------+
| char_length | varchar_length | text_length |
+-------------+----------------+-------------+
|        28   |           28   |        28   |
+-------------+----------------+-------------+
1 row in set (0.00 sec)
```

varchar 和 text 列的长度与预期一样，但读者或许会认为 char 列长度应该为 30，因为上文已提到过 char 列中存放的字符串是使用空格向右补齐的。MySQL 服务器在获取数据时会删除 char 类型数据的尾端空格，因此无论存储字符串的列为何种类型，该字符串函数得到的结果都是相同的。

除了确定字符串的长度，或许还需要查找字符串中子字符串的位置。举例来说，如果希望查找字符串‘characters’在 vchar_fld 列中的位置，可以使用 position()函数，如下所示。

```
mysql> SELECT POSITION('characters' IN vchar_fld)
    -> FROM string_tbl;
+------------------------------------+
| POSITION('characters' IN vchar_fld) |
+------------------------------------+
|                                 19  |
+------------------------------------+
1 row in set (0.12 sec)
```

如果找不到该子字符串，那么 position()函数将返回 0。

警告

对于 C 或 C++程序员来说，数组的第一个元素的位置号为 0，但必须记住在使用数据库时，字条串中的第一字符位置号为 1。因此如果 position()函数返回 0 值则表示没有找到该子字符串，而不是该子字符串出现在字符串的第一位置。

如果希望在字符串中的任意位置开始搜索，而不是仅限于从第一个字符开始，那么可以使用 locate()函数，它与 position()函数相似，只不过它可以接受可选的第三个参数，该参数用于指定搜索的起始位置。Locate()vchar_fld 列中第 5 个字符之后查找字符串'is'出现的位置：

```
mysq1>SHLECT LOCATE('is', vchar_fld, 5)
    ->FROM string_tbl;
+----------------------------+
| LOCATE('is', vchar_fld, 5)  |
+----------------------------+
|                        13   |
+----------------------------+
1 row in set (0.02 sec)
```

提示

Oracle 数据库并不包含 position()或 locate()函数，但是可以用它的 instr()函数模仿 position()函数（需提供两个参数）和 locate()函数（需提供 3 个参数）。SQL Server 也没有 position()或 locate()函数，但是它的 charindx()函数与 Orcale 的 instr()函数相似，同样可以接受 2 个或 3 个参数。

另一个接受字符串作为参数同时返回数字的函数是字符串比较函数 strcmp()。只有 MySQL 实现了 strcmp()，并且在 Oracle 及 SQL Server 数据库中并没有与此功能相似的函数。该函数接受两个字符串作为参数，并返回下面的结果之一：

- −1，第一个字符串的排序位于第二个字符串之前；
- 0，两个字符串相同；
- 1，第一个字符串的排序位于第二个字符串之后。

为了说明该函数是如何工作的，下面首先使用一个查询显示 5 个字符串的次序，然后显示使用 strcmp()将一个字符串与另一个字符串比较的结果。下面为插入到 string_tbl 表的 5 个字符串：

```
mysql> DELETE FROM string_tbl;
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('abcd');
Query OK, 1 row affected (0.03 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('xyz');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('QRSTUV');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('qrstuv');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('12345');
Query OK, 1 row affected (0.00 sec)
```

下面为 5 个字符串的排序次序：

```
mysql> SELECT vchar_fld
    -> FROM string_tbl
    -> ORDER BY vchar_fld;
+-----------+
| vchar_fld |
+-----------+
| 12345     |
| abcd      |
| QRSTUV    |
| qrstuv    |
| xyz       |
+-----------+
5 rows in set (0.00 sec)
```

下面的查询对 5 个不同字符串进行 6 次比较：

```
mysql> SELECT STRCMP('12345','12345') 12345_12345,
    ->  STRCMP('abcd','xyz') abcd_xyz,
    ->  STRCMP('abcd','QRSTUV') abcd_QRSTUV,
    ->  STRCMP('qrstuv','QRSTUV') qrstuv_QRSTUV,
    ->  STRCMP('12345','xyz') 12345_xyz,
    ->  STRCMP('xyz','qrstuv') xyz_qrstuv;
+-----------+--------+-----------+-------------+---------+----------+
|12345_12345|abcd_xyz|abcd_QRSTUV|qrstuv_QRSTUV|12345_xyz|xyz_qrstuv|
+-----------+--------+-----------+-------------+---------+----------+
|      0    |      -1|       -1  |          0  |       -1|      1   |
+-----------+--------+-----------+-------------+---------+----------+
1 row in set (0.00 sec)
```

显然第一个比较的结果为 0，因为它是字符串与自身的比较。第 4 个比较的结果也是 0，这个结果令人惊讶，因为尽管这两个字符串是由相同的字母组成，但其中一个字符串都是大写，而另一个为小写。原因在于 MySQL 的 strcmp()函数是大小写不敏感的，在使用此函数时必须记住这一点。另外 4 个比较操作返回-1 或 1，取决于第一个字符串在排序次序中位于第二个字符串之前还是之后。举例来说，strcmp('abcd', 'xyz')产生-1，因为字符串‘abcd’出现在字符串‘xyz’之前。

除了 strcmp()函数，MySQL 还可以在 select 语句中使用 like 和 regexp 操作符来比较字符串。这些比较操作的结果为 1（true）或 0（false）。因此，这些操作符同样可以构建表达式并返回字符串，与上一节中描述的函数十分相似。下面是使用 like 的一个例子：

```
mysql> SELECT name, name LIKE '%ns' ends_in_ns
    -> FROM department;
+----------------+------------+
| name           | ends_in_ns |
+----------------+------------+
| Operations     |          1 |
| Loans          |          1 |
| Administration |          0 |
+----------------+------------+
3 rows in set (0.25 sec)
```

上例获取了所有的部门名称以及一个表达式结果。其中，当部门名称以“ns”结尾时返回 1；反之，返回 0。希望执行更复杂的模式匹配时，可以使用 regexp 操作符，例如：

```
mysql> SELECT cust_id, cust_type_cd, fed_id,
    ->   fed_id REGEXP '.{3}-.{2}-.{4}' is_ss_no_format
    -> FROM customer;
+---------+--------------+-------------+-----------------+
| cust_id | cust_type_cd | fed_id      | is_ss_no_format |
+---------+--------------+-------------+-----------------+
|       1 | I            | 111-11-1111 |               1 |
|       2 | I            | 222-22-2222 |               1 |
|       3 | I            | 333-33-3333 |               1 |
|       4 | I            | 444-44-4444 |               1 |
|       5 | I            | 555-55-5555 |               1 |
|       6 | I            | 666-66-6666 |               1 |
|       7 | I            | 777-77-7777 |               1 |
|       8 | I            | 888-88-8888 |               1 |
|       9 | I            | 999-99-9999 |               1 |
|      10 | B            | 04-1111111  |               0 |
|      11 | B            | 04-2222222  |               0 |
|      12 | B            | 04-3333333  |               0 |
|      13 | B            | 04-4444444  |               0 |
+---------+--------------+-------------+-----------------+
13 rows in set (0.00 sec)
```

在本查询中，如果 fed_id 列的值与社会安全号码的格式相匹配，第四列就将返回 1。

> **提示**
> SQL Server 和 Oracle 数据库的用户可以使用 case 表达式获得相似的结果，在第 11 章中对此进行详细描述。

返回字符串的字符串函数

在某些情况下，需要修改已有的字符串，比如截取其中的一部分，或者向它添加额外的文本。每种数据库服务器都包含多个执行这些任务的函数，不过在开始介绍它们之前需要再次重设 string_tbl 表的数据。

```
mysql> DELETE FROM string_tbl;
Query OK, 5 rows affected (0.00 sec)

mysql> INSERT INTO string_tbl (text_fld)
    -> VALUES ('This string was 29 characters');
Query OK, 1 row affected (0.01 sec)
```

本章之前已经介绍了如何使用 concat()函数来构建包含重音符号的字符串，concat()函数在许多其他场合也是十分有用的，包括向已存储的字符串后附加额外的字符。例如，下面的例子修改了 text_fld 列存储的字符串，在其末尾增加了一个短句。

```
mysql> UPDATE string_tbl
    -> SET text_fld = CONCAT(text_fld, ', but now it is longer');
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

text_fld 列的内容现在变成如下所示：

```
mysql> SELECT text_fld
    -> FROM string_tbl;
+------------------------------------------------------+
| text_fld                                             |
+------------------------------------------------------+
| This string was 29 characters, but now it is longer |
+------------------------------------------------------+
1 row in set (0.00 sec)
```

由此可见，与其他返回字符串的函数一样，可以使用 concat()来替换字符列所存储的数据。

另一种使用 concat()函数的常用方式是根据独立的数据片段构建字符串。举例来说，下面的查询将为每个银行柜员产生简介字符串：

```
mysql> SELECT CONCAT(fname, ' ', lname, ' has been a ',
    ->   title, ' since ', start_date) emp_narrative
    -> FROM employee
    -> WHERE title = 'Teller' OR title = 'Head Teller';
+----------------------------------------------------------+
| emp_narrative                                            |
+----------------------------------------------------------+
| Helen Fleming has been a Head Teller since 2008-03-17   |
| Chris Tucker has been a Teller since 2008-09-15         |
| Sarah Parker has been a Teller since 2006-12-02         |
```

```
| Jane Grossman has been a Teller since 2006-05-03          |
| Paula Roberts has been a Head Teller since 2006-07-27     |
| Thomas Ziegler has been a Teller since 2004-10-23         |
| Samantha Jameson has been a Teller since 2007-01-08       |
| John Blake has been a Head Teller since 2004-05-11        |
| Cindy Mason has been a Teller since 2006-08-09            |
| Frank Portman has been a Teller since 2007-04-01          |
| Theresa Markham has been a Head Teller since 2005-03-15 |
| Beth Fowler has been a Teller since 2006-06-29            |
| Rick Tulman has been a Teller since 2006-12-12            |
+-----------------------------------------------------------+
13 rows in set (0.30 sec)
```

concat()函数可以处理返回字符串的任何表达式，甚至可以将数字和日期型转换为字符串格式，如上面作为参数的日期列（start_date）。尽管 Oracle 数据库也包含了 concat()函数，但是它只能接受两个字符串参数，因此前面的查询不能运行在 Oracle 下。作为替代品，可以使用连接操作符（||）而不是函数调用，如下所示。

```
SELECT fname || ' ' || lname || ' has been a ' ||
  title || ' since ' || start_date emp_narrative
FROM employee
WHERE title = 'Teller' OR title = 'Head Teller';
```

SQL Server 中并没有 concat()函数，因此需要使用与上述查询相同的方法，只不过使用的是 SQL Server 的连接操作符（+）或不是||。

concat()函数可以用于在字符串首端或末端添加字符，除此之外，或许还需要在字符串中间增加或替换部分字符。上面 3 种数据库都为此功能提供了不同的实现函数，下面首先介绍 MySQL 中的相应函数，然后再展示其他两种数据库服务器的函数。

MySQL 包含了 insert()函数，它接受 4 个参数：原始字符串、字符串操作的开始位置、需要替换的字符数以及替换字符串。根据第三个参数值，函数可以选择插入或替换原始字符串中的字符。如果该参数值为 0，那么替换字符串将会被插入其中，并且剩余的字符将会向右排放。

```
mysql> SELECT INSERT('goodbye world', 9, 0, 'cruel ') string;
+---------------------+
| string              |
+---------------------+
| goodbye cruel world |
+---------------------+
1 row in set (0.00 sec)
```

在本例中，所有从第 9 位开始的字符都向右移动，以容纳插入的字符串‘cruel’。如果第三个参数大于 0，那么相应数目的字符将会被替换字符串所取代。

```
mysql> SELECT INSERT('goodbye world', 1, 7, 'hello') string;
+-------------+
| string      |
+-------------+
```

```
| hello world |
+-------------+
1 row in set (0.00 sec)
```

在本例中，前 7 个字符被字符串‘hello’替换。Oracle 数据并没有提供与 MySQL 的 insert()函数具有同样灵活性的函数，但它的 replace()函数也可以用于替换子字符串。下面使用 replace()重新完成上一个例子：

```
SELECT REPLACE('goodbye world', 'goodbye', 'hello')
FROM dual;
```

其中，所有子字符串‘goodbye’都被字符串‘hello’所替换，从而产生结果为‘hello world’。replace()函数将使用替换字符串取代搜索字符串的所有实例，因此，必须小心它可能进行了多次替换操作，超出了原先的预计。

SQL Server 同样包含 replace()函数，其功能与 Oracle 的相同。除此之外，SQL Server 还包含了 stuff()函数，其功能与 MySQL 的 insert()函数类似，例如：

```
SELECT STUFF('hello world', 1, 5, 'goodbye cruel')
```

在执行该查询后，从位置 1 开始的 5 个字符被删除，然后‘goodbye cruel’被插入到该位置，从而产生的结果字符串为‘goodby cruel world’。

除了向字符串中插入字符外，或许还需要从字符串中提取子字符串。对于此项功能，3 种数据库服务器都提供了 substring()函数（Oracle 数据库中为 substr()），它从指定的位置开始提取指定数目的字符。下面的例子演示如何从字符串的第 9 个位置提取 5 个字符：

```
mysql> SELECT SUBSTRING('goodbye cruel world', 9, 5);
+---------------------------------------+
| SUBSTRING('goodbye cruel world', 9, 5) |
+---------------------------------------+
| cruel                                  |
+---------------------------------------+
1 row in set (0.00 sec)
```

除了本节介绍的这些函数，3 种服务器还内建了大量处理字符串数据的其他函数。它们之中大多数用于非常特定的目的，比如产生 8 进制或 16 进制数的字符串，但也有许多通用的字符串函数，比如去除或补齐字符串末端的空格。建议读者通过阅读所使用数据库服务器的用户手册或更通用的 SQL 用户手册（如 *SQL in a Nutshell*）来获取更详细的信息。

7.2 使用数值数据

与字符串数据（以及后面将要介绍的临时数据）不同，数值型数据的生成十分简单，可以通过输入一个数字、从另一个数值列提取或者通过计算而产生。所有的算术操作符（+、-、*、/）都可以用于执行计算，并且可以使用括号改变优先级，如下所示。

```
mysql> SELECT (37 * 59) / (78 - (8 * 6));
+---------------------------+
| (37 * 59) / (78 - (8 * 6)) |
+---------------------------+
|                  72.77    |
+---------------------------+
1 row in set (0.00 sec)
```

正如在第 2 章中提到的，如果数值型数据的精度大于所在列的指定长度，那么在其被存储时可能会发生取整操作。例如，数字 9.96 在被存放到定义为 float(3,1)的列时将会被取整为 10.0。

7.2.1 执行算术函数

大多数内建的处理数值的函数都用于特定的算术目的，比如求出某数的平方根。表 7-1 列出了一些常用的数值函数，它们接受单个参数并返回一个数字。

表 7-1 单参数数值函数

函 数 名	描 述
Acos(*x*)	Calculates the arc cosine of *x*
Asin(*x*)	Calculates the arc sine of *x*
Atan(*x*)	Calculates the arc tangent of *x*
Cos(*x*)	Calculates the cosine of *x*
Cot(*x*)	Calculates the cotangent of *x*
Exp(*x*)	Calculates e^x
Ln(*x*)	Calculates the natural log of *x*
Sin(*x*)	Calculates the sine of *x*
Sqrt(*x*)	Calculates the square root of *x*
Tan(*x*)	Calculates the tangent of *x*

这些函数执行特定的任务，但本书不打算为它们提供相关用例（因为从函数名和描述就可以轻易得知它们的用途，如果哪个函数你还不能确定，那么很可能你并不需要使用它）。其他数值函数多用于计算，其灵活性稍高一点，因此下面进行一些解释。

举例来说，用于计算两数相除所得余数的求模操作在 MySQL 和 Oracle 数据库中都是通过 mod()函数实现的。下面的例子计算 4 除 10 的余数：

```
mysql> SELECT MOD(10,4);
+-----------+
| MOD(10,4) |
+-----------+
|         2 |
+-----------+
1 row in set (0.02 sec)
```

mod()函数主要用于整型参数，但 MySQL 还可以用它来处理实数，例如：

```
mysql> SELECT MOD(22.75, 5);
+---------------+
| MOD(22.75, 5) |
+---------------+
|        2.75   |
+---------------+
1 row in set (0.02 sec)
```

提示

SQL Server 并没有 mod()函数。作为替代品，它使用操作符%来计算余数。表达式 10 % 4 产生的结果为 2。

另一种接受两个数值参数的函数是 pow()（如果使用 Oracle 或 SQL Server 数据库则为 power()），它返回两个参数的幂计算，即求第一个参数的第二个参数幂次方，如下所示。

```
mysql> SELECT POW(2,8);
+----------+
| POW(2,8) |
+----------+
|     256  |
+----------+
1 row in set (0.03 sec)
```

由此可见，MySQL 的 pow(2,8)用于计算 2^8。因为计算机内存通常是由 2^x 个字节为单位分配的，因此 pow()函数可以作为一种获取某段内存确切字节数的简易方法。

```
mysql> SELECT POW(2,10) kilobyte, POW(2,20) megabyte,
    ->  POW(2,30) gigabyte, POW(2,40) terabyte;
+----------+----------+------------+---------------+
| kilobyte | megabyte | gigabyte   | terabyte      |
+----------+----------+------------+---------------+
|    1024  |  1048576 | 1073741824 | 1099511627776 |
+----------+----------+------------+---------------+
1 row in set (0.00 sec)
```

虽然不知道读者的看法如何，但显然记住 1GB 等于 2^{30}B 比记住数字 1 073 741 824 更为容易。

7.2.2 控制数字精度

当处理浮点数时，或许不需要总是以数字的全部精度进行计算及显示。举例来说，或许以 6 位数字的精度存储金融交易数据是合适的，但一般在显示时精确到百位数就可以了。有 4 个函数可用于限制浮点数的精度，即 ceil()、floor()、round()和 truncate()。所有 3 种数据库都包含这些函数，只不过 Oracle 数据库中用 trunc()替代 truncate()，SQL

Server 使用 ceiling()替代 ceil()。

ceil()和 floor()函数用于向上或向下截取整型数字，例如：

```
mysql> SELECT CEIL(72.445), FLOOR(72.445);
+--------------+---------------+
| CEIL(72.445) | FLOOR(72.445) |
+--------------+---------------+
|         73   |          72   |
+--------------+---------------+
1 row in set (0.06 sec)
```

可见，72 和 73 之间的任何数都会被 ceil()函数取为 73，而 floor()函数的结果为 72。需要记住的是 ceil()函数向上进位，无论该数字的小数部分是多么小，floor()则将向下进位，即使其小数部分比较大，如下所示。

```
mysql> SELECT CEIL(72.000000001), FLOOR(72.999999999);
+--------------------+---------------------+
| CEIL(72.000000001) | FLOOR(72.999999999) |
+--------------------+---------------------+
|             73     |               72    |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

如果在应用中使用它们不合适，那么可以使用 round()函数来向上或向下取整，其结果取决于两个整数之间的中间点（即四舍五入），例如：

```
mysql> SELECT ROUND(72.49999), ROUND(72.5), ROUND(72.50001);
+-----------------+-------------+-----------------+
| ROUND(72.49999) | ROUND(72.5) | ROUND(72.50001) |
+-----------------+-------------+-----------------+
|           72    |        73   |            73   |
+-----------------+-------------+-----------------+
1 row in set (0.00 sec)
```

使用 round()，对于小数部分大于等于 0.5 的数值将向上取整，反之则向下取整。

大多数时候，需要按照特定位数保留小数部分，而不是对它取整，因此 round()函数还提供了一个可选的第二个参数以指定在小数点右侧保留多少位。下一个例子显示了如何利用第二个参数将数字 72.0909 保留 1 位、2 位或 3 位小数：

```
mysql> SELECT ROUND(72.0909, 1), ROUND(72.0909, 2), ROUND(72.0909, 3);
+-------------------+-------------------+-------------------+
| ROUND(72.0909, 1) | ROUND(72.0909, 2) | ROUND(72.0909, 3) |
+-------------------+-------------------+-------------------+
|           72.1    |           72.09   |           72.091  |
+-------------------+-------------------+-------------------+
1 row in set (0.00 sec)
```

与 round()函数类似，truncate()函数也允许接受第二个可选参数，以指定小数点右侧保

留的位数，但 truncate()只是简单地去掉不需要的小数位，而不进行四舍五入。下例显示了如何将数字 72.0909 截取为分别带有 1 位、2 位或 3 位小数的数字：

```
mysql> SELECT TRUNCATE(72.0909, 1), TRUNCATE(72.0909, 2),
    ->  TRUNCATE(72.0909, 3);
+--------------------+---------------------+--------------------+
|TRUNCATE(72.0909, 1)| TRUNCATE(72.0909, 2) | TRUNCATE(72.0909,3) |
+--------------------+---------------------+--------------------+
|              72.0  |             72.09   |            72.090  |
+--------------------+---------------------+--------------------+
1 row in set (0.00 sec)
```

提示

SQL Server 并不包含 truncate()函数，作为替代品，它的 round()函数可以接受第三个可选参数，如果提供了该参数并且非 0，那么将对数字进行截取操作而不是取整操作。

Truncate()和 round()函数都可以为第二个参数指定一个负数，表示小数点左侧需要被截取或取整多少位，看起来似乎有些奇怪，但在实际中可能是有用的。举例来说，如果你只能以 10 为单位来售卖产品，某个顾客订购了 17 个单位，那么需要在下面的做法中进行选择，以修改顾客的订单数量：

```
mysql> SELECT ROUND(17, -1), TRUNCATE(17, -1);
+---------------+-----------------+
| ROUND(17, -1) | TRUNCATE(17, -1) |
+---------------+-----------------+
|          20   |            10   |
+---------------+-----------------+
1 row in set (0.00 sec)
```

如果例子中的产品是图钉，那么向订购 17 个图钉的顾客卖出 10 个还是 20 个图钉并没有太大区别，但如果卖的是劳力士手表，显然取整操作是更好的做法。

7.2.3 处理有符号数

如果所用数字列允许存储负数（在第 2 章中已经展示了如何将数字列声明为 unsigned，即该列只允许存放正数），那么可能会用到下面几种数值函数。例如，在生成每个银行账户当前状态的报表时，可以使用下面的包含 3 列结果的查询：

```
mysql> SELECT account_id, SIGN(avail_balance), ABS(avail_balance)
    -> FROM account;
+------------+---------------------+--------------------+
| account_id | SIGN(avail_balance) | ABS(avail_balance) |
+------------+---------------------+--------------------+
|        1   |                  1  |          1057.75   |
|        2   |                  1  |           500.00   |
|        3   |                  1  |          3000.00   |
|        4   |                  1  |          2258.02   |
```

```
|      5    |                 1    |           200.00   |
| ...                                                   |
|     19    |                 1    |          1500.00   |
|     20    |                 1    |         23575.12   |
|     21    |                 0    |             0.00   |
|     22    |                 1    |          9345.55   |
|     23    |                 1    |         38552.05   |
|     24    |                 1    |         50000.00   |
+-----------+----------------------+--------------------+
24 rows in set (0.00 sec)
```

其中，第二列使用了 sign()函数，它在账户余额为负数时返回-1，在账户余额为 0 时返回 0，而在账户余额为正数时返回 1。第三列使用 abs()函数返回账户余额的绝对值。

7.3 使用时间数据

本章讨论 3 种数据类型（字符型、数值型和时间型），其中时间数据的生成和操作最为复杂，造成其复杂性的部分原因在于存在众多记录日期和时间的方式。例如，作者写作本段内容的日期可以使用下面多种方式表示：

- Wednesday, September 17, 2008；
- 9/17/2008 2:14:56 P.M. EST；
- 9/17/2008 19:14:56 GMT；
- 2612008 (Julian format)；
- Star date [−4] 85712.03 14:14:56 (*Star Trek* format)。

这些差别中的一小部分仅仅是格式上的问题，但其复杂性大多与所在时区有关，下面将进行详细的描述。

7.3.1 处理时区

世界各地的人们都将太阳直射本地的时间作为正午，因此没有办法强迫所有人使用统一的时钟。因此，世界被划分为 24 个时区，同一时区内部的所有人都参照相同的时钟，而不同时区的人们则使用不同的时钟。这看起来很简单，但考虑到有些地区在 1 年中会两次调整他们的时钟（即实行夏时制），而有些地区却不采取这种做法，这样会造成地球上的某两个地方在 1 年中有半年的时差为 4 小时，而另外半年的时差为 5 小时，如此大大增加了复杂性。即使在同一时区的各地区，由于有的实行夏时制，有的不实行，也会造成在一年中有半年的时间是相同的，另外半年却有 1 个小时的差距。

人们从大航海时代开始就需要处理时差问题，到计算机时代则加剧了这一问题的严重性。为了保证参照的时间是统一的，15 世纪的航海家们将他们的时钟设置为英国格林

威治时间，即著名的格林威治标准时间或 GMT。所有其他的时区都可以使用与 GMT 所差距的小时数来表述。举例来说，对于美国东部的时区，即东部标注时间可以用 GMT -5:00 来表示，或者说比 GMT 早 5 个小时。

现在，还经常使用 GMT 的一个变种，即协调世界时，或称之为 UTC，它基于原子时钟（更精确的说法是，在世界范围内 50 个地点的 200 个原子周期的平均时间，因此被称为世界时）。SQL Server 和 MySQL 都提供了返回当前 UTC 时间戳的函数（SQL Server 中为 getutcdate()，MySQL 中为 utc_timestamp()）。

大多数数据库服务器根据当前所在地区设置默认的时区，并提供工具以便在需要的时候进行修改。举个例子，如果数据库用于存储世界范围内的股票交易，通常就会配置为 UTC 时间；如果数据库用于存储特定零售企业的销售数据，那么通常会使用服务器所在时区的时间。

MySQL 提供两个不同的时区设置：全局时区和会话时区，后者可能对于每个登录用户都是不同的。可以通过下面的查询查看这两种设置：

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
+--------------------+---------------------+
| @@global.time_zone | @@session.time_zone |
+--------------------+---------------------+
| SYSTEM             | SYSTEM              |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

结果值为 system，这表明服务器根据数据库所在地使用相应的时区设置。

如果你正坐在瑞士苏黎世的一台计算机前，并且通过网络打开了通向位于纽约的一台 MySQL 服务器的会话，那么可以通过执行下面的命令改变当前会话的时区设置：

```
mysql> SET time_zone = 'Europe/Zurich';
Query OK, 0 rows affected (0.18 sec)
```

再次检查时区，将会看到下面的结果：

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
+--------------------+---------------------+
| @@global.time_zone | @@session.time_zone |
+--------------------+---------------------+
| SYSTEM             | Europe/Zurich       |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

此时在会话中显示的所有日期都符合苏黎世时间。

提示

Oracle 数据库的用户可以使用下面的命令修改会话的时区设置：

```
ALTER SESSION TIMEZONE = 'Europe/Zurich'
```

7.3.2 生成时间数据

可以使用下面任意一种方法产生时间数据：

- 从已有的 date、datetime 或 time 列中复制数据；
- 执行返回 date、datetime 或 time 型数据的内建函数；
- 构建可以被服务器识别的代表日期的字符串。

为了使用最后一种方法，必须首先理解格式化日期的各种组件。

表示日期数据的字符串

在第 2 章中的表 2-6 中提供了较常用的日期部件，而为了方便读者加深记忆，表 7-2 显示了同样的日期部件。

载入 MySQL 时区数据

如果你是在 Windows 平台上运行 MySQL 服务器，那么可以在设置全局或会话时区之前以手动方式载入时区数据，这需要进行下面几步操作:

1. 在 http://dev.mysql.com/downloads/timezones.html 上下载时区数据;
2. 关闭 MySQL 服务器;
3. 从下载的 ZIP 文件（本书在下载时得到的文件为 timezone-2006p.zip）中提取所有文件，并放到 MySQL 的安装目录下的/data/mysql（本书中使用的完整路径为/Program Files/MySQL/MySQL Server 6.0/data/mysql）中;
4. 重新启动 MySQL 服务器。

如果需要查看时区数据，必须首先使用 use mysql 切换到系统自带的 mysql 数据库，并执行下面的查询:

```
mysql> SELECT name FROM time_zone_name;
+----------------------------------+
| name                             |
+----------------------------------+
| Africa/Abidjan                   |
| Africa/Accra                     |
| Africa/Addis_Ababa               |
| Africa/Algiers                   |
| Africa/Asmera                    |
| Africa/Bamako                    |
| Africa/Bangui                    |
| Africa/Banjul                    |
| Africa/Bissau                    |
| Africa/Blantyre                  |
| Africa/Brazzaville               |
```

```
| Africa/Bujumbura                   |
...
| US/Alaska                          |
| US/Aleutian                        |
| US/Arizona                         |
| US/Central                         |
| US/East-Indiana                    |
| US/Eastern                         |
| US/Hawaii                          |
| US/Indiana-Starke                  |
| US/Michigan                        |
| US/Mountain                        |
| US/Pacific                         |
| US/Pacific-New                     |
| US/Samoa                           |
| UTC                                |
| W-SU                               |
| WET                                |
| Zulu                               |
+------------------------------------+
546 rows in set (0.01 sec)
```

可以使用本表中与所在地区相匹配的时区名称来修改 MySQL 服务器的时区设置。

表 7-2　数据格式组件

组　件	定　义	范　围
YYYY	年份,包括世纪	1000～9999
MM	月份	01 (January)～12 (December)
DD	日	01～31
HH	小时	00～23
HHH	小时(过去的)	−838～838
MI	分钟	00～59
SS	秒	00～59

为了构建服务器，可以将之识别为 date、datetime 或 time 类型的字符串，需要按照表 7-3 中所显示的顺序来整合各种日期部件。

表 7-3　必需的日期部件

类　型	默认格式
Date	YYYY-MM-DD
Datetime	YYYY-MM-DD HH:MI:SS
Timestamp	YYYY-MM-DD HH:MI:SS
Time	HHH:MI:SS

因此，为了向 datetime 列中添加一条 2008 年 9 月 17 日下午 15:00 点的时间数据，需要使用下面的字符串：

```
'2008-09-17 15:30:00'
```

如果服务器需要接受 datetime 型的数据值，比如更新某个 datetime 列或者调用接受 datetime 参数的内建函数，那么可以为之提供一个按照必需的日期部件构建的具有正确格式的字符串，服务器将自动进行转换。例如，下面的语句用于修改银行交易的日期：

```
UPDATE transaction
SET txn_date = '2008-09-17 15:30:00'
WHERE txn_id = 99999;
```

因为 set 子句中的字符串将被适配到 datetime 列，所以服务器必须首先确定该字符串是否为 datetime 类型的值。因此，服务器将试着通过将字符串分解为 datetime 格式所默认的 6 个组件（年、月、日、小时、分钟、秒钟）的方式，对其进行转换。

字符串到日期的转换

如果服务器并没有期望 datetime 类型的值，或者使用了非默认格式来表示 datetime，那么就需要告知服务器将字符串转换为 datetime。例如，下面的简单查询中使用了 cast() 函数返回了 datetime 类型的值。

```
mysql> SELECT CAST('2008-09-17 15:30:00' AS DATETIME);
+-----------------------------------------+
| CAST('2008-09-17 15:30:00' AS DATETIME) |
+-----------------------------------------+
| 2008-09-17 15:30:00                     |
+-----------------------------------------+
1 row in set (0.00 sec)
```

本章末尾将详细介绍 cast()函数，这里的例子主要展示如何构建 datetime 类型的值，而对于 date 和 time 类型也可以使用同样的逻辑。下面的查询使用 cast()函数产生了 date 和 time 类型的值：

```
mysql> SELECT CAST('2008-09-17' AS DATE) date_field,
    ->  CAST('108:17:57' AS TIME) time_field;
+------------+------------+
| date_field | time_field |
+------------+------------+
| 2008-09-17 | 108:17:57  |
+------------+------------+
1 row in set (0.00 sec)
```

当然，也可以在服务器需要 date、datetime 或 time 值时显式地转换字符串，这样服务器将不会进行隐式转换。

无论是显式或是隐式的，在字符串被转换为时间值时，必须按照规定次序提供所有的日期部件。某些服务器对日期格式要求十分严格，而 MySQL 服务器对于各部件之间的

分隔符的要求却比较宽松。例如，MySQL 可以接受下面各种表示 2008 年 9 月 17 日下午 15:30 的字符串：

```
'2008-09-17 15:30:00'
'2008/09/17 15:30:00'
'2008,09,17,15,30,00'
'20080917153000'
```

尽管这样带来了一定的灵活性，但如果不使用默认日期部件来产生时间值仍是不允许的，下面展示的内建函数将具有比 cast()函数高得多的灵活性。

产生日期的函数

如果需要根据字符串产生时间数据，但所提供的字符串不是 cast()函数所接受的格式，那么可以使用内建函数将字符串格式化为日期字符串，比如 MySQL 包含的 str_to_date()函数。例如，从文件中获取了字符串‘September 17, 2008’并用它来更新日期列，显然该字符串并非需要的 YYYY-MM-DD 格式，但是可以使用 str_to_date()格式化，使之能用于 cast()函数：

```
UPDATE individual
SET birth_date = STR_TO_DATE('September 17, 2008', '%M %d, %Y')
WHERE cust_id = 9999;
```

str_to_date()函数的第二个参数指明了待转换字符串的日期格式，本例中%M 代表月，%d 代表天，而%Y 代表 4 位数字的年份。该函数可识别 30 多种格式部件（format component），表 7-4 定义了最常用的一些部件。

表 7-4　日期格式部件

格 式 部 件	描　　述
%M	月名称（1 月～12 月）
%m	月序号（01～12）
%d	日序号（01～31）
%j	日在一年中的序号（001～366）
%W	星期名称（星期日～星期六）
%Y	4 位数字表示的年份
%y	两位数字表示的年份
%H	小时（00～23）
%h	小时（01～12）
%i	分钟（00～59）
%s	秒钟（00～59）
%f	微秒（000000～999999）
%p	A.M.或 P.M.

str_to_date()函数将根据格式字符串的内容返回 datetime、date 或 time 类型值。举例来说，如果格式字符串只包含%H、%i 或%s，那么将返回 time 值。

提示

Oracle 数据库的用户可以使用 to_date()函数执行与 MySQL 的 str_to_date()函数同样的功能。SQL Server 包含的 convert()函数提供的灵活性不及 MySQL 和 Oracle 数据库，它只能接受 21 种预定义格式的日期字符串，而没有提供可定制的日期格式构件。

如果需要产生当前日期/时间，那么不必手动构造字符串，而是直接利用内建函数获取系统时钟并返回当前的日期或时间字符串：

```
mysql> SELECT CURRENT_DATE(), CURRENT_TIME(), CURRENT_TIMESTAMP();
+----------------+----------------+---------------------+
| CURRENT_DATE() | CURRENT_TIME() | CURRENT_TIMESTAMP() |
+----------------+----------------+---------------------+
| 2008-09-18     | 19:53:12       | 2008-09-18 19:53:12 |
+----------------+----------------+---------------------+
1 row in set (0.12 sec)
```

这些函数将按照所返回时间类型的默认格式返回当前日期/时间值。Oracle 数据库包含 current_date()和 current_timestamp()函数，但没有 current_time()，SQL Server 则只包含 current_timestamp()函数。

7.3.3 操作时间数据

本小节将对接受日期参数，同时返回日期、字符串或数字的内建函数进行说明。

返回日期的时间函数

许多内建的时间函数接受一个日期型值作为参数，然后返回另一个日期。例如，MySQL 的 date_add()函数可以为指定日期增加任意一段时间间隔（如天、月、年）并产生另一个日期。下面的例子展示了如何为当前日期增加 5 天：

```
mysql> SELECT DATE_ADD(CURRENT_DATE(), INTERVAL 5 DAY);
+------------------------------------------+
| DATE_ADD(CURRENT_DATE(), INTERVAL 5 DAY) |
+------------------------------------------+
| 2008-09-22                               |
+------------------------------------------+
1 row in set (0.06 sec)
```

其中，第二个参数包含了 3 个元素：interval 关键字、所需要增加的数量以及时间间隔的类型。表 7-5 显示了一些常用的时间间隔类型。

表 7-5　常用的时间间隔类型

间 隔 名 称	描　　述
Second	秒数
Minute	分钟数
Hour	小时数
Day	天数
Month	月份
Year	年份
Minute_second	分钟数和秒数，中间用“:”隔开
Hour_second	小时数、分钟数和秒数，中间用“:”隔开
Year_month	年份和月份，中间用“-”隔开

表 7-5 中的前 6 种类型是简单易懂的，而后 3 种类型包含了多个元素，需要进一步解释。例如，你得知 ID 为 9999 的交易实际发生的时间比 transaction 表中当前记录的时间要晚 3 小时 27 分钟 11 秒，那么可以使用下面的方法进行修正：

```
UPDATE transaction
SET txn_date = DATE_ADD(txn_date, INTERVAL '3:27:11' HOUR_SECOND)
WHERE txn_id = 9999;
```

在此例中，date_add()函数获取 txn_date 列的值，并向其增加 3 小时 27 分 11 秒，然后使用结果值修改 txn_date 列。

或者你在人力资源部工作，并发现 ID 为 4789 的员工所填报的年龄比他的实际年龄大，因而需要向他的出生日期增加 9 年 11 个月，那么可以执行下面的语句：

```
UPDATE employee
SET birth_date = DATE_ADD(birth_date, INTERVAL '9-11' YEAR_MONTH)
WHERE emp_id = 4789;
```

提示

SQL Server 可以使用 dateadd()函数实现上一个例子:

```
UPDATE employee
SET birth_date =
  DATEADD(MONTH, 119, birth_date)
WHERE emp_id = 4789
```

SQL Server 不能使用复合的时间间隔（如 year_month），因此需要首先将 9 年 11 个月转换成 119 个月。

Oracle 数据库用户可以使用 add_months()函数实现这个例子:

```
UPDATE employee
SET birth_date = ADD_MONTHS(birth_date, 119)
WHERE emp_id = 4789;
```

在某些情况下，或许你需要向某个日期增加一段时间间隔，不过你只知道目标时间却不清楚它离原来的日期差距的具体天数。假设某个银行顾客登录网上银行并定制月底的账目移交。可以通过编写一些代码求得当前的月份并计算到月底剩余的天数，但更好的方法是调用 last_day()函数，它可以替你完成这些工作（MySQL 和 Oracle 数据库都包含 last_day()函数， SQL Server 没有提供与之功能相似的函数）。如果顾客在 2008 年 9 月 17 日发出移交请求，那么可以使用下面的方法求得当月的最后一天：

```
mysql> SELECT LAST_DAY('2008-09-17');
+------------------------+
| LAST_DAY('2008-09-17') |
+------------------------+
| 2008-09-30             |
+------------------------+
1 row in set (0.10 sec)
```

无论所提供的参数是 date 型还是 datetime 型，last_day()函数都将返回一个 date 值。尽管该函数表面上并没有节省大量时间，但实际上它在底层处理了相当烦琐的逻辑，比如在需要求出二月份的最后一天时必须首先确定当前年度是否为闰年。

另一个返回 date 的函数能够将某个时区的 datetime 值转换为另一个时区对应的时间。在 MySQL 中，该函数为 convert_tz()，在 Oracle 数据库中为 new_time()函数。例如，需要将当前本地时间转换为 UTC 时间，那么可以进行下面的查询：

```
mysql> SELECT CURRENT_TIMESTAMP() current_est,
    ->   CONVERT_TZ(CURRENT_TIMESTAMP(), 'US/Eastern', 'UTC') current_utc;
+---------------------+---------------------+
| current_est         | current_utc         |
+---------------------+---------------------+
| 2008-09-18 20:01:25 | 2008-09-19 00:01:25 |
+---------------------+---------------------+
1 row in set (0.76 sec)
```

在收到与数据库所在地不同的另一个时区的日期值时，可以利用该函数进行转换。

返回字符串的时间函数

返回字符串的时间函数大多数用于提取日期或时间的某一部分。例如，MySQL 包含的 dayname()函数可以确定某一日期是星期几：

```
mysql> SELECT DAYNAME('2008-09-18');
+-----------------------+
| DAYNAME('2008-09-18') |
+-----------------------+
| Thursday              |
+-----------------------+
1 row in set (0.08 sec)
```

MySQL 中的这类函数多用于提取日期值中的信息，但本书建议使用 extract()函数实现此目的，因为记住一个函数的几种变体比记住一系列不同的函数更容易一些。此外，extract()函数还是 SQL:2003 标准的一部分，并且在 Oracle 数据库中同样得到了实现。

extract()函数使用与 date_add()函数相同的时间间隔类型（参见表 7-5）来定义日期中所感兴趣的元素。例如，需要提取 datetime 值中的年份，可以执行如下操作：

```
mysql> SELECT EXTRACT(YEAR FROM '2008-09-18 22:19:05');
+------------------------------------------+
| EXTRACT(YEAR FROM '2008-09-18 22:19:05') |
+------------------------------------------+
|                                     2008 |
+------------------------------------------+
1 row in set (0.00 sec)
```

提示

SQL Server 没有包含 extract()函数，但提供了 datepart()函数。下面演示如何使用 datepart()函数来提取 datetime 值中的年份:

```
SELECT DATEPART(YEAR, GETDATE())
```

返回数字的时间函数

本章前面介绍了向某个日期值增加一段时间间隔并产生另一个日期值的函数，而另一种常见的行为是接受两个日期值，并求出它们之间的时间间隔（天、星期或年）。为了实现此操作，mysql 提供了函数 datediff()，它返回两个日期之间的天数。例如，我想知道自己孩子的暑假一共有多少天，可以作如下查询：

```
mysql> SELECT DATEDIFF('2009-09-03', '2009-06-24');
+--------------------------------------+
| DATEDIFF('2009-09-03', '2009-06-24') |
+--------------------------------------+
|                                   71 |
+--------------------------------------+
1 row in set (0.05 sec)
```

看来在孩子安全返回学校之前的 71 天内，我不得不忍受他在家大闹天宫似的折腾了。datediff()函数忽略了参数中的时钟值，即使将前一个日期的时钟设置为一天的最后一秒，而将第二个日期的时钟设为从午夜开始的第一秒，也不会对计算结果造成影响：

```
mysql> SELECT DATEDIFF('2009-09-03 23:59:59', '2009-06-24 00:00:01');
+--------------------------------------------------------+
| DATEDIFF('2009-09-03 23:59:59', '2009-06-24 00:00:01') |
+--------------------------------------------------------+
|                                                     71 |
+--------------------------------------------------------+
1 row in set (0.00 sec)
```

如果交换两个参数的次序，将较早的日期作为第一个参数，那么函数将返回负值：

```
mysql> SELECT DATEDIFF('2009-06-24', '2009-09-03');
+--------------------------------------+
| DATEDIFF('2009-06-24', '2009-09-03') |
+--------------------------------------+
|                                  -71 |
+--------------------------------------+
1 row in set (0.01 sec)
```

提示

SQL Server 同样包含 datediff()函数，但比 MySQL 中的实现更具灵活性，即可以为其指定时间间隔的类型（如年、月、日、小时等），而不是只能计算两个日期之间的天数。下面给出 SQL Server 中上一个例子的查询实现方法：

```
SELECT DATEDIFF(DAY, '2009-06-24', '2009-09-03')
```

此外，Oracle 数据库允许通过两个日期值相减的方式求出它们间隔的天数。

7.4 转换函数

本章前面已经介绍了如何使用 cast()函数将字符串转换为 datetime 值。事实上每种数据库服务器都提供了不少用于转换数据类型的专有函数，但本书推荐使用 cast()函数，它属于 SQL:2003 标准，并且在 MySQL、Oracle 和 Microsoft SQL Server 中均被实现。

使用 cast()时必须提供一个作为关键字的值或表达式，以及所需要转换的类型。下面是将字符串转换为整数的例子：

```
mysql> SELECT CAST('1456328' AS SIGNED INTEGER);
+-----------------------------------+
| CAST('1456328' AS SIGNED INTEGER) |
+-----------------------------------+
|                           1456328 |
+-----------------------------------+
1 row in set (0.01 sec)
```

当 cast()函数在将字符串转换为数字时，首先会从左向右试着对整个字符串进行转换，如果期间在字符串中遇到非数字的字符，那么转换将中止并且不返回错误。考虑下面的例子：

```
mysql> SELECT CAST('999ABC111' AS UNSIGNED INTEGER);
+---------------------------------------+
| CAST('999ABC111' AS UNSIGNED INTEGER) |
+---------------------------------------+
```

```
|                                    999 |
+----------------------------------------+
1 row in set, 1 warning (0.08 sec)
mysql> show warnings;
+---------+------+------------------------------------------------+
| Level   | Code | Message                                        |
+---------+------+------------------------------------------------+
| Warning | 1292 | Truncated incorrect INTEGER value: '999ABC111' |
+---------+------+------------------------------------------------+
1 row in set (0.07 sec)
```

在本例中，字符串的前 3 个数字被成功转换，剩余部分则被忽略，因而产生的结果值为 999。不过服务器同时会产生一个警告，以提示字符串并没有被完全转换。

如果需要将字符串转换为 date、time 或 datetime 类型的值，就必须严格遵守每种类型的默认格式。如果待转换的日期字符串不是默认格式（比如 datetime 类型为 YYYY-MM-DD HH:MI:SS），那么首先需要使用其他函数将之重新排列，比如本章前面介绍的 MySQL 的 str_to_date()函数。

7.5 小测验

下面的练习用于测试读者对本章阐述的内建函数的理解，答案参见附录 C。

练习 7-1

编写查询，返回字符串‘Please find the substring in this string’的第 17 个和第 25 个字符。

练习 7-2

编写查询，返回数字-25.76823 的绝对值与符号（-1、0 或 1），并将返回值四舍五入至百分位。

练习 7-3

编写查询，返回当前日期所在的月份。

第 8 章

分组与聚集

对数据库用户来说，通常数据的存储粒度总是越低越好。如果在银行查账时需要检查每个客户交易，就需要在数据库中存储独立的交易，但这并不意味着用户必须按数据在数据库中的存储方式对其进行处理。本章聚焦于如何对数据进行分组与聚集，以使用户在更高的粒度层次上与数据进行交互。

8.1 分组概念

有时需要在数据中找到变化的趋势，这就需要数据库服务器在产生所需要的结果集之前对数据进行一些加工。举例来说，假设你掌管了银行的业务操作，或许你会想知道每个柜员创建了多少个账户。下面首先列出对原始数据的查询：

```
mysql> SELECT open_emp_id
    -> FROM account;
+-------------+
| open_emp_id |
+-------------+
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
```

```
|        10   |
|        13   |
|        13   |
|        13   |
|        16   |
|        16   |
|        16   |
|        16   |
|        16   |
|        16   |
+-------------+
24 rows in set (0.01 sec)
```

account 表中只有 24 行数据，很容易看出它们分别被 4 个不同的职员创建，其中 ID 为 16 的职员创建了 6 个账户。但是，如果该银行具有大量职员和成千上万个账户，那么此查询结果将十分繁杂而难以查看。

因此，这时可以使用 group by 子句请求数据库服务器对数据进行分组，下面是同样的查询，只是根据职员 ID 使用 group by 子句对账户数据进行了分组：

```
mysql> SELECT open_emp_id
    -> FROM account
    -> GROUP BY open_emp_id;
+-------------+
| open_emp_id |
+-------------+
|         1   |
|        10   |
|        13   |
|        16   |
+-------------+
4 rows in set (0.00 sec)
```

结果集中每行对应了一个 open_emp_id 列的独立值，因此一共只有 4 行，而不是全部 24 行。结果集变小的原因在于 4 个职员每个人都创建了不只一个账户。如果想要查到每个柜员所创建的账户数，那么可以在 select 子句中使用聚集函数，以统计每个分组的行数目：

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|         1   |       8  |
|        10   |       7  |
|        13   |       3  |
|        16   |       6  |
+-------------+----------+
4 rows in set (0.00 sec)
```

其中，聚集函数 count()计算了每个分组的行数，星号表示对分组的所有列计数。通过联合使用 group by 子句和 count()聚集函数，可以在不查看原始数据的情况下，精确地满足业务问题对数据的需要。

当对数据分组时，或许还需要在结果集中过滤掉不想要的数据，并且过滤条件是针对分组数据而不是原始数据。由于 group by 子句在 where 子句被评估之后运行，因此无法对 where 子句增加过滤条件。例如，下面的查询视图过滤掉创建账户小于 5 个的职员：

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> WHERE COUNT(*) > 4
    -> GROUP BY open_emp_id;
ERROR 1111 (HY000): Invalid use of group function
```

上例错在不应该在 where 子句中使用聚集函数 count(*)，因为在评估 where 子句时分组还未被创建，所以必须在 having 子句中使用分组过滤条件。下面展示了使用 having 后的查询结果：

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> HAVING COUNT(*) > 4;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          16 |        6 |
+-------------+----------+
3 rows in set (0.00 sec)
```

通过 having 子句，那些不足 5 个账户数的分组数据被过滤掉，所以结果集中只包含创建了 5 个或更多账户的职员，即从结果集中除去了 ID 为 13 的职员。

8.2 聚集函数

聚集函数对某个分组的所有行执行特定的操作。尽管每种数据库服务器都具有独有的聚集函数，但一些通用的聚集函数在所有主流服务器上都得到了实现。

Max()

返回集合中的最大值。

Min()

返回集合中的最小值。

Avg()

返回集合中的平均值。

Sum()

返回集合中所有值的和。

Count()

返回集合中值的个数。

下面的查询使用各种通用聚集函数来分析所有核算账户（checking accounts）的可用余额：

```
mysql> SELECT MAX(avail_balance) max_balance,
    ->  MIN(avail_balance) min_balance,
    ->  AVG(avail_balance) avg_balance,
    ->  SUM(avail_balance) tot_balance,
    ->  COUNT(*) num_accounts
    -> FROM account
    -> WHERE product_cd = 'CHK';
+-----------+-----------+-----------+----------+-----------+
|max_balance|min_balance |avg_balance |tot_balance |num_accounts |
+-----------+-----------+-----------+----------+-----------+
|   38552.05 |     122.37 |7300.800985 |   73008.01 |         10 |
+-----------+-----------+-----------+----------+-----------+
1 row in set (0.09 sec)
```

该查询的结果指出，在 account 表中共有 10 个核算账户，最大余额为$38 552.05，最小余额为$122.37，平均余额为$7 300.80，10 个账户的余额总和为$73 008.01。本例主要是为了帮助读者对聚集函数有个大致的印象，下面将进一步阐明如何使用这些函数。

8.2.1 隐式或显式分组

在上一个例子中，查询返回的每个值都是由聚集函数产生的，这些聚集函数作用于使用过滤条件 product_cd = 'CHK'指定的分组上的所有行。这里没有使用 group by 子句，因此它是一个隐式分组（即包含查询返回的所有行）。

不过在大多数情况下，除了聚集函数所产生的列外，还需要获取额外的列。比如说，假设你需要扩展前一个查询，使之为每种产品类型执行同样的 5 种聚集函数，而不是只针对核算账户。对此查询来说，需要在列举 5 个聚集函数结果的同时提取 product_cd 列：

```
SELECT product_cd,
  MAX(avail_balance) max_balance,
```

```
  MIN(avail_balance) min_balance,
  AVG(avail_balance) avg_balance,
  SUM(avail_balance) tot_balance,
  COUNT(*) num_accounts
FROM account;
```

然而，在执行此查询时，将会收到下面的错误：

```
ERROR 1140 (42000): Mixing of GROUP columns (MIN(),MAX(),COUNT(),...)
with no GROUP
columns is illegal if there is no GROUP BY clause
```

虽然查询中似乎很明显地指定要获取account表中每种产品的集合并对它们使用聚集函数，但是由于没有显式地指定如何对数据分组而导致查询失败。因此，需要为它增加一个 group by 子句以指定聚集函数所作用行的分组：

```
mysql> SELECT product_cd,
    ->  MAX(avail_balance) max_balance,
    ->  MIN(avail_balance) min_balance,
    ->  AVG(avail_balance) avg_balance,
    ->  SUM(avail_balance) tot_balance,
    ->  COUNT(*) num_accts
    -> FROM account
    -> GROUP BY product_cd;
+---------+---------+---------+----------+---------+---------+
|product_cd |max_balance |min_balance |avg_balance  |tot_balance|num_accts |
+---------+---------+---------+----------+---------+---------+
| BUS      | 9345.55   |  0.00     |4672.774902 |  9345.55 |    2     |
| CD       | 10000.00  | 1500.00   |4875.000000 | 19500.00 |    4     |
| CHK      | 38552.05  |  122.37   |7300.800985 | 73008.01 |   10     |
| MM       | 9345.55   | 2212.50   |5681.713216 | 17045.14 |    3     |
| SAV      |  767.77   |  200.00   | 463.940002 |  1855.76 |    4     |
| SBL      | 50000.00  | 50000.00  |50000.000000| 50000.00 |    1     |
+---------+---------+---------+----------+---------+---------+
6 rows in set (0.00 sec)
```

通过 group by 子句，服务器将为在 product_cd 列上具有同样值的行产生分组，然后在这 6 个分组上应用 5 种聚集函数。

8.2.2 对独立值计数

当使用 count()函数确定每个分组成员数目时，可以选择是对分组中所有成员计数还是只计数某个列的不同值。例如，考虑下面的数据，即为每个账户开户的相应雇员信息：

```
mysql> SELECT account_id, open_emp_id
    -> FROM account
    -> ORDER BY open_emp_id;
+------------+------------+
| account_id |open_emp_id  |
+------------+------------+
```

```
|        8   |          1   |
|        9   |          1   |
|       10   |          1   |
|       12   |          1   |
|       13   |          1   |
|       17   |          1   |
|       18   |          1   |
|       19   |          1   |
|        1   |         10   |
|        2   |         10   |
|        3   |         10   |
|        4   |         10   |
|        5   |         10   |
|       14   |         10   |
|       22   |         10   |
|        6   |         13   |
|        7   |         13   |
|       24   |         13   |
|       11   |         16   |
|       15   |         16   |
|       16   |         16   |
|       20   |         16   |
|       21   |         16   |
|       23   |         16   |
+------------+------------+
24 rows in set (0.00 sec)
```

如上所示，这些账户的开户者只包括 4 个不同的雇员（ID 为 1、10、13 和 16）。假设需要创建查询以获取完成开户的雇员数，而不是通过观察结果来手动计数，那么将 count()函数应用到 open_emd_id 列就会看到下面的结果：

```
mysql> SELECT COUNT(open_emp_id)
    -> FROM account;
+--------------------+
| COUNT(open_emp_id) |
+--------------------+
|             24    |
+--------------------+
1 row in set (0.00 sec)
```

在此情况下，指定 open_emp_id 列作为计数列所产生的结果与指定 count(*)相同。如果希望对分组的不同值计数而不是统计分组的所有行，则需要指定 distinct 参数，如下所示：

```
mysql> SELECT COUNT(DISTINCT open_emp_id)
    -> FROM account;
+-----------------------------+
| COUNT(DISTINCT open_emp_id) |
```

```
+-----------------------------+
|                      4      |
+-----------------------------+
1 row in set (0.00 sec)
```

通过指定 distinct，count()函数检查分组每个成员的特定列的值，并去除发生重复的行，而不是简单地对分组中所有行进行计数。

8.2.3 使用表达式

除了使用列作为聚集函数的参数外，还可以创建表达式作为参数。例如，想要找到所有账户中 pending deposit 值（即 pending balance 减去 available balance）的最大值，则可以使用下面的查询：

```
mysql> SELECT MAX(pending_balance - avail_balance) max_uncleared
    -> FROM account;
+---------------+
| max_uncleared |
+---------------+
|      660.00   |
+---------------+
1 row in set (0.00 sec)
```

本例中使用的表达式是相当简单的，但实际上用于聚集函数的参数表达式可以根据需要任意增加复杂度，只需要保证最后返回一个数字、字符串或日期即可。在第 11 章中将说明如何在聚集函数中使用 case 表达式，以确定特定的行是否需要被包含到某一聚集中。

8.2.4 如何处理 null 值

当执行聚集函数或其他数值计算时，应当首先考虑 null 值是否可能影响计算结果。下面对此进行说明，首先需要构建一个简单的数值型数据表，并使用集合{1,3,5}对其初始化：

```
mysql> CREATE TABLE number_tbl
    ->  (val SMALLINT);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO number_tbl VALUES (1);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (3);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (5);
Query OK, 1 row affected (0.00 sec)
```

考虑下面的查询，它针对该数字集合执行 5 个聚集函数：

```
mysql> SELECT COUNT(*) num_rows,
    ->  COUNT(val) num_vals,
    ->  SUM(val) total,
    ->  MAX(val) max_val,
    ->  AVG(val) avg_val
    -> FROM number_tbl;
+----------+----------+-------+---------+---------+
| num_rows | num_vals | total | max_val | avg_val |
+----------+----------+-------+---------+---------+
|       3  |       3  |    9  |      5  | 3.0000  |
+----------+----------+-------+---------+---------+
1 row in set (0.08 sec)
```

结果与预料的一样，count(*)和 count(val)返回的值均为 3，sum(val)返回 9，max(val)返回 5，avg(val)返回 3。下一步将向 number_tbl 表中添加一个 null 值并再次运行查询：

```
mysql> INSERT INTO number_tbl VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT COUNT(*) num_rows,
    ->  COUNT(val) num_vals,
    ->  SUM(val) total,
    ->  MAX(val) max_val,
    ->  AVG(val) avg_val
    -> FROM number_tbl;
+----------+----------+-------+---------+---------+
| num_rows | num_vals | total | max_val | avg_val |
+----------+----------+-------+---------+---------+
|       4  |       3  |    9  |      5  | 3.0000  |
+----------+----------+-------+---------+---------+
1 row in set (0.00 sec)
```

即使表中增加了一个 null 值，sum()、max()和 avg()函数的返回值也没有发生变化，这表明它们忽略了任何遇到的 null 值。count(*)函数返回值为 4，这是由于 number_tbl 表现在包含了 4 行，而 count(val)函数仍旧返回 3。它们的区别在于 count(*)对行的数目计数，而 count(val)对 val 列所包含的值的数目进行计数并且忽略所有遇到的 null 值。

8.3　产生分组

通常人们很少对原始数据感兴趣，而更希望对原始数据进行加工以便适应数据分析的需要。常见的数据加工的例子包括：

- 产生某个区域的合计数，比如欧洲市场的销售额；
- 发现极端值，比如 2005 年业绩最佳的销售员；
- 确定某事重复出现的频率，比如每个支行的新开户数。

为了解决这些问题，需要请求数据库服务器根据列（一个或多个）或表达式对数据行进行分组。正如前面几个例子中所演示的那样，可以在查询中使用 group by 子句作为分组数据的方法。本节将说明如何根据一个或多个列进行数据分组，如何使用表达式分组数据，以及如何在各分组中产生合计数。

8.3.1 对单列的分组

对单列的分组是最简单同时也是最常用的。例如，想要找到每种产品的余额总计，可以根据 account.product_cd 列来分组：

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> GROUP BY product_cd;
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| BUS        |     9345.55  |
| CD         |    19500.00  |
| CHK        |    73008.01  |
| MM         |     17045.14 |
| SAV        |     1855.76  |
| SBL        |    50000.00  |
+------------+--------------+
6 rows in set (0.00 sec)
```

该查询产生 6 个分组，分别对应每种产品，然后对每个分组成员的可用余额进行合计。

8.3.2 对多列的分组

在某些情况下，需要根据多列产生分组。下面扩展上一个例子，假设需要查找的不是每种产品的余额合计，而是同时根据产品和开户支行进行统计（比如说，在 Woburn 支行所开户的核算账户的余额总计是多少）。下面的例子显示了如何完成此任务：

```
mysql> SELECT product_cd, open_branch_id,
    ->  SUM(avail_balance) tot_balance
    -> FROM account
    -> GROUP BY product_cd, open_branch_id;
+------------+----------------+-------------+
| product_cd | open_branch_id | tot_balance |
+------------+----------------+-------------+
| BUS        |              2 |    9345.55  |
| BUS        |              4 |       0.00  |
| CD         |              1 |   11500.00  |
| CD         |              2 |    8000.00  |
| CHK        |              1 |     782.16  |
| CHK        |              2 |    3315.77  |
| CHK        |              3 |    1057.75  |
| CHK        |              4 |   67852.33  |
```

```
| MM         |              1 |    14832.64 |
| MM         |              3 |     2212.50 |
| SAV        |              1 |      767.77 |
| SAV        |              2 |      700.00 |
| SAV        |              4 |      387.99 |
| SBL        |              3 |    50000.00 |
+------------+----------------+-------------+
14 rows in set (0.00 sec)
```

该查询产生了 14 个分组，分别对应 account 表中每种产品与支行的组合。因为 open_branch_id 是从表中获取而不能由聚集函数产生，所以必须同时在 select 子句和 group by 子句中增加此列。

8.3.3 利用表达式分组

除了根据列分组数据，还可以根据表达式产生的值进行分组。考虑下面的查询，它根据职员入职年份对职员分组：

```
mysql> SELECT EXTRACT(YEAR FROM start_date) year,
    ->  COUNT(*) how_many
    -> FROM employee
    -> GROUP BY EXTRACT(YEAR FROM start_date);
+------+----------+
| year | how_many |
+------+----------+
| 2004 |        2 |
| 2005 |        3 |
| 2006 |        8 |
| 2007 |        3 |
| 2008 |        2 |
+------+----------+
5 rows in set (0.15 sec)
```

该查询使用的表达式十分简单，只是使用 extract()函数获取并返回日期中的年份，然后据此对 employee 表的数据行进行分组。

8.3.4 产生合计数

在 8.3.2 小节中，已经举例说明了如何为每种产品与支行的组合产生账户余额合计数。但现在假设需要在为每种产品/支行组合计算合计余额的同时，还需要为每种产品单独计算合计数。可以采取的方法包括增加一个附加查询并将结果合并到一起，或将查询结果载入电子表格中进行二次统计，或是构建 Perl 脚本、Java 程序以及其他方法来获取数据并执行附加的计算。不过更好的办法是使用 with rollup 选项来请求数据库服务器完成这些事。下面是在 group by 子句中使用 with rollup 修改后的查询：

```
mysql> SELECT product_cd, open_branch_id,
    ->  SUM(avail_balance) tot_balance
    -> FROM account
```

```
    -> GROUP BY product_cd, open_branch_id WITH ROLLUP;
+------------+----------------+-------------+
| product_cd | open_branch_id | tot_balance |
+------------+----------------+-------------+
| BUS        |              2 |     9345.55 |
| BUS        |              4 |        0.00 |
| BUS        |           NULL |     9345.55 |
| CD         |              1 |    11500.00 |
| CD         |              2 |     8000.00 |
| CD         |           NULL |    19500.00 |
| CHK        |              1 |      782.16 |
| CHK        |              2 |     3315.77 |
| CHK        |              3 |     1057.75 |
| CHK        |              4 |    67852.33 |
| CHK        |           NULL |    73008.01 |
| MM         |              1 |    14832.64 |
| MM         |              3 |     2212.50 |
| MM         |           NULL |    17045.14 |
| SAV        |              1 |      767.77 |
| SAV        |              2 |      700.00 |
| SAV        |              4 |      387.99 |
| SAV        |           NULL |     1855.76 |
| SBL        |              3 |    50000.00 |
| SBL        |           NULL |    50000.00 |
| NULL       |           NULL |   170754.46 |
+------------+----------------+-------------+
21 rows in set (0.02 sec)
```

现在在结果集中有 7 个额外的行，分别对应 6 个独立的产品以及总合计数（所有产品的合计）。对于 6 种产品的合计行，其中 open_branch_id 列为 null，因为这些合计是对所有支行进行计算的。例如，观察输出结果的第 3 行，将会看到 BUS 账户在所有支行的余额总计为$9 345.55。对于最后一行的总合计数，其 product_cd 和 open_branch_id 列都为 null，它显示所有产品和支行的余额总计为$170 754.46。

提示

如果使用 Oracle 数据库，需要使用不同的语法来指明执行合计操作。前面的查询在 Oracle 数据库中的 group by 子句应如下所示：

```
GROUP BY ROLLUP(product_cd, open_branch_id)
```

该语法的优点在于允许在 group by 子句中对某个子集执行合计操作。例如，需要服务器根据列 a、b 和 c 进行分组，但只需要根据 b 和 c 进行合计，那么可以使用下面的做法：

```
GROUP BY a, ROLLUP(b, c)
```

如果处理计算产品合计，还需要为每个支行计算合计，那么可以使用 with cube 选项，它可以为分组列所有可能的组合产生合计行。不幸的是，MySQL 6.0 版并不支持 with cube，但在 SQL Server 和 Oracle 数据中是可用的。下面是使用 with cube 的一个例子，

其中去掉了 mysql>提示符以表明该查询还不能在 MySQL 中执行：

```
SELECT product_cd, open_branch_id,
  SUM(avail_balance) tot_balance
FROM account
GROUP BY product_cd, open_branch_id WITH CUBE;
+------------+----------------+-------------+
| product_cd | open_branch_id | tot_balance |
+------------+----------------+-------------+
| NULL       |           NULL |   170754.46 |
| NULL       |              1 |    27882.57 |
| NULL       |              2 |    21361.32 |
| NULL       |              3 |    53270.25 |
| NULL       |              4 |    68240.32 |
| BUS        |              2 |     9345.55 |
| BUS        |              4 |        0.00 |
| BUS        |           NULL |     9345.55 |
| CD         |              1 |    11500.00 |
| CD         |              2 |     8000.00 |
| CD         |           NULL |    19500.00 |
| CHK        |              1 |      782.16 |
| CHK        |              2 |     3315.77 |
| CHK        |              3 |     1057.75 |
| CHK        |              4 |    67852.33 |
| CHK        |           NULL |    73008.01 |
| MM         |              1 |    14832.64 |
| MM         |              3 |     2212.50 |
| MM         |           NULL |    17045.14 |
| SAV        |              1 |      767.77 |
| SAV        |              2 |      700.00 |
| SAV        |              4 |      387.99 |
| SAV        |           NULL |     1855.76 |
| SBL        |              3 |    50000.00 |
| SBL        |           NULL |    50000.00 |
+------------+----------------+-------------+
25 rows in set (0.02 sec)
```

使用 with cube 的查询比使用 with rollup 多产生了 4 行，分别对应 4 个支行的合计数。与 with rollup 类似，其中 product_cd 的值被设为 null，以表示该合计是针对支行执行的。

提示

同样地，如果使用 Oracle 数据库执行 cube 操作，也需要修改一下语法，上一个查询的 group by 子句应为：

```
GROUP BY CUBE(product_cd, open_branch_id)
```

8.4 分组过滤条件

在第 4 章中已经介绍了过滤条件的各种类型，并演示了如何在 where 子句中使用它们。

当分组数据时，也可以在产生分组后对数据应用过滤条件，having 子句就是放置这类过滤条件的地方。考虑下面的例子：

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    -> GROUP BY product_cd
    -> HAVING SUM(avail_balance) >= 10000;
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| CD         |     19500.00 |
| CHK        |     73008.01 |
| MM         |     17045.14 |
| SBL        |     50000.00 |
+------------+--------------+
4 rows in set (0.00 sec)
```

该查询包含两个过滤条件：一个在 where 子句中，它用于过滤掉不活动的账户；另一个在 having 子句中，它过滤掉可用余额合计小于$10 000 的产品。因此，第一个过滤条件在分组之前执行，第二个过滤条件则在分组产生以后才作用于数据。如果错误地将两个过滤条件都放到 where 子句中，将会产生下面的错误：

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    ->   AND SUM(avail_balance) > 10000
    -> GROUP BY product_cd;
ERROR 1111 (HY000): Invalid use of group function
```

因为查询的 where 子句中不能包含聚集函数，所以该查询失败。这是因为 where 子句是在分组之前被评估的，因此服务器此时还不能对分组执行任何函数。

警告

当在包含 group by 子句的查询中增加过滤条件时，需要仔细考虑过滤是针对原始数据（此时过滤条件应放到 where 子句中），还是针对分组后的数据（此时过滤条件应放到 having 子句中）。

此外，还可以在 having 子句中包含未在 select 语句中出现的聚集函数，例如：

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    -> GROUP BY product_cd
    -> HAVING MIN(avail_balance) >= 1000
    ->   AND MAX(avail_balance) <= 10000;
+------------+--------------+
| product_cd | prod_balance |
```

```
+------------+-------------+
| CD         |    19500.00 |
| MM         |    17045.14 |
+------------+-------------+
2 rows in set (0.00 sec)
```

该查询为每种活动的产品产生余额总计，但在 having 子句中根据过滤条件排除所有最小余额低于$1 000 或最大余额大于$10 000 的产品。

8.5 小测验

完成下面的练习以掌握 SQL 的分组与聚集特性。在附录 C 中可以找到相应的解答。

练习 8-1

构建查询，对 account 表的数据行计数。

练习 8-2

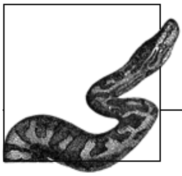
修改练习 8-1 中的查询，使之对每个客户所持有的账户计数，并且显示每个客户的 ID 及其账户数。

练习 8-3

修改练习 8-2 的查询，使之只包含至少持有两个账户的客户。

练习 8-4（附加题）

查找至少包含一个账户的产品和支行组合的可用余额合计数，并根据余额合计数对结果进行排序（从最高到最低）。



第 9 章 子查询

子查询是一种可用于总共 4 种 SQL 语句的强大工具。本章将详细探讨子查询的多种应用。

9.1 什么是子查询

子查询是指包含在另一个 SQL 语句（下文称包含语句）内部的查询。子查询总是由括号包围，并且通常在包含语句之前执行。像其他查询一样，子查询也会返回一个如下类型的结果集：

- 单列单行；
- 单列多行；
- 多列多行。

子查询返回的结果集类型决定了它可能如何被使用以及包含语句可能使用哪些运算符来处理子查询返回的数据。任何子查询返回的数据在包含语句执行完成后都会被丢弃，这使子查询像一个具有作用域的临时表（这就意味着服务器在 SQL 语句执行结束后将清空子查询结果所占的内存）。

事实上，读者已经在前面的章节中看到了许多子查询的例子，不过现在还是以一个简单的例子开始：

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE account_id = (SELECT MAX(account_id) FROM account);
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|       29   | SBL        |      13 |      50000.00 |
```

```
+------------+-----------+---------+--------------+
1 row in set (0.65 sec)
```

在这个例子中，子查询返回 account 表中 account_id 列的最大值，包含语句则返回对应账户的相关数据。如果还不清楚子查询是如何运行的，那么读者可以单独运行子查询（不包含括弧）来看看它返回什么。下面是上一个例子的子查询：

```
mysql> SELECT MAX(account_id) FROM account;
+-----------------+
| MAX(account_id) |
+-----------------+
|              29 |
+-----------------+
1 row in set (0.00 sec)
```

子查询返回单列单行的结果，因此它可以被用作等式条件中的其中一个表达式（如果子查询返回的结果多于 1 行，它可以被用于比较，而不能用于等式判断，后续章节将会介绍）。在这种情况下，读者可以先获得子查询返回的值，然后用它替换包含查询过滤条件中的右边表达式，例如：

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE account_id = 29;
+------------+-----------+---------+--------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+-----------+---------+--------------+
|         29 | SBL       |      13 |     50000.00 |
+------------+-----------+---------+--------------+
1 row in set (0.02 sec)
```

在上述情况中，子查询很有用，因为它允许读者使用单一查询检索最大编号账号的相关信息，而不是首先使用一个查询获取最大 account_id，然后由另一个查询从该 account 表中获取所需的数据。正如读者将会了解的，子查询在许多其他情况下也是有用的，并可能成为读者的 SQL 工具包中最强大的工具之一。

9.2 子查询类型

除了前面探讨过的子查询结果集类型之外（单行/单列，单行/多列或者多行多列），还可以基于另外一个因素划分子查询：一些子查询完全独立（称为非关联子查询），其他的则引用包含语句中的列（称为关联查询）。

9.3 非关联子查询

前面章节中的例子都是非关联查询，它可以单独执行而不需要引用包含语句中的任何

内容。当然读者遇到的大多数子查询都是这种类型，但是更新或者删除语句会经常用到关联子查询（后面的章节会继续讲述）。前面的子查询除了是非关联的外，返回的都是一个单行单列的表。这种类型的子查询称为标量子查询，并且可以位于常用运算符（=、<>、<、>、<=、>=）的任意一边。下面的例子向读者展示如何在不等条件中使用标量子查询：

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE open_emp_id <> (SELECT e.emp_id
    ->   FROM employee e INNER JOIN branch b
    ->     ON e.assigned_branch_id = b.branch_id
    ->   WHERE e.title = 'Head Teller' AND b.city = 'Woburn');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|          7 | CHK        |       3 |       1057.75 |
|          8 | MM         |       3 |       2212.50 |
|         10 | CHK        |       4 |        534.12 |
|         11 | SAV        |       4 |        767.77 |
|         12 | MM         |       4 |       5487.09 |
|         13 | CHK        |       5 |       2237.97 |
|         14 | CHK        |       6 |        122.37 |
|         15 | CD         |       6 |      10000.00 |
|         18 | CHK        |       8 |       3487.19 |
|         19 | SAV        |       8 |        387.99 |
|         21 | CHK        |       9 |        125.67 |
|         22 | MM         |       9 |       9345.55 |
|         23 | CD         |       9 |       1500.00 |
|         24 | CHK        |      10 |      23575.12 |
|         25 | BUS        |      10 |          0.00 |
|         28 | CHK        |      12 |      38552.05 |
|         29 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
17 rows in set (0.86 sec)
```

这个查询返回所有不是由 Woburn 分行的总柜台员开户的账户的相关数据（子查询基于每个分行只有一个总柜台员这个假设）。本例中的子查询连接了两个表，而且包括两个过滤条件，因此要比前面的复杂一点。子查询可以利用各种可用查询子句（select、from、where、 group by、 having 和 order by），读者可以或简单或复杂地使用它。

如果在等式条件下使用子查询，而子查询又返回多行结果，那么将会出错。例如，修改前面的查询以使子查询返回 Woburn 分行所有的柜台员而不是仅有一个的总柜台员 ，那么就会出现如下所示的错误：

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE open_emp_id <> (SELECT e.emp_id
```

```
    ->   FROM employee e INNER JOIN branch b
    ->     ON e.assigned_branch_id = b.branch_id
    ->   WHERE e.title = 'Teller' AND b.city = 'Woburn');
ERROR 1242 (21000): Subquery returns more than 1 row
```

单独运行子查询，结果如下：

```
mysql> SELECT e.emp_id
    -> FROM employee e INNER JOIN branch b
    ->   ON e.assigned_branch_id = b.branch_id
    -> WHERE e.title = 'Teller' AND b.city = 'Woburn';
+--------+
| emp_id |
+--------+
|     11 |
|     12 |
+--------+
2 rows in set (0.02 sec)
```

包含查询失败是因为表达式 open_emp_id 不能等于表达式集(包含 emp_id 为 11 和 12 的集合)。换句话说，单一事物不能等于多个事物的集合。下面将介绍如何使用另一个不同的运算符解决这个问题。

9.3.1 多行单列子查询

正如前面的例子所说明的，返回多行结果的子查询不能在等式条件的一边使用。不过，另外 4 个运算符可以用来为这些类型的子查询构建条件。

in 和 not in 运算符

虽然不能把一个值与一个值集进行相等比较，但是可以检查一个值集中能否包含某一个值。下面的例子没有使用子查询，但可以说明如何使用 in 这个运算符构建条件在值集中查找一个值。

```
mysql> SELECT branch_id, name, city
    -> FROM branch
    -> WHERE name IN ('Headquarters', 'Quincy Branch');
+-----------+---------------+---------+
| branch_id | name          | city    |
+-----------+---------------+---------+
|         1 | Headquarters  | Waltham |
|         3 | Quincy Branch | Quincy  |
+-----------+---------------+---------+
2 rows in set (0.03 sec)
```

条件左边的表达式是 name 列，而右边是字符的集合。运算符 in 的作用是检查 name 列值中是否有两个字串中任意一个：如果有，则条件满足，该行也将被加入到结果集。读者可以使用两个等式条件实现：

```
mysql> SELECT branch_id, name, city
    -> FROM branch
    -> WHERE name = 'Headquarters' OR name = 'Quincy Branch';
+-----------+---------------+---------+
| branch_id | name          | city    |
+-----------+---------------+---------+
|         1 | Headquarters  | Waltham |
|         3 | Quincy Branch | Quincy  |
+-----------+---------------+---------+
2 rows in set (0.01 sec)
```

当集合只包含两个表达式时，这种方法似乎是合理的，但是如果集合包含许多（或者成百上千的）值，那么很容易理解使用 in 运算符更为合适。

读者可能会为条件的一边创建一个字符串、日期或数字的集合，但更可能通过执行子查询产生这个集合。下面的查询利用 in 运算符在右边的过滤条件中构建子查询以查找哪些雇员是主管。

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id IN (SELECT superior_emp_id
    ->   FROM employee);
+--------+---------+-----------+--------------------+
| emp_id | fname   | lname     | title              |
+--------+---------+-----------+--------------------+
|      1 | Michael | Smith     | President          |
|      3 | Robert  | Tyler     | Treasurer          |
|      4 | Susan   | Hawthorne | Operations Manager |
|      6 | Helen   | Fleming   | Head Teller        |
|     10 | Paula   | Roberts   | Head Teller        |
|     13 | John    | Blake     | Head Teller        |
|     16 | Theresa | Markham   | Head Teller        |
+--------+---------+-----------+--------------------+
7 rows in set (0.01 sec)
```

子查询返回所有主管的 ID，包含查询则从 employee 表检索这些雇员的 4 列值。

```
mysql> SELECT superior_emp_id
-> FROM employee;
+-----------------+
| superior_emp_id |
+-----------------+
|            NULL |
|               1 |
|               1 |
|               3 |
|               4 |
|               4 |
|               4 |
```

```
|                4 |
|                4 |
|                6 |
|                6 |
|                6 |
|               10 |
|               10 |
|               13 |
|               13 |
|               16 |
|               16 |
+------------------+
18 rows in set (0.00 sec)
```

正如读者看到的，有些雇员的 ID 被列出了不止一次，这是由于他们同时管理多个人。不过这并不会影响到包含查询的结果，因为一个雇员的 ID 在子查询的结果集中出现一次还是多次没有差别。当然，读者如果对子查询返回重复项感到困扰，那么可以在子查询的 select 子句中增加关键字 distinct，并且不会改变包含查询的结果集。

除了判断一个值集中是否包含某一个值外，读者还可以使用 not in 运算符检查相反情况，即是否不包含。下面的例子是前面查询的另一种版本，它用 not in 替换了 in：

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id NOT IN (SELECT superior_emp_id
    ->   FROM employee
    ->   WHERE superior_emp_id IS NOT NULL);
+--------+----------+----------+----------------+
| emp_id | fname    | lname    | title          |
+--------+----------+----------+----------------+
|      2 | Susan    | Barker   | Vice President |
|      5 | John     | Gooding  | Loan Manager   |
|      7 | Chris    | Tucker   | Teller         |
|      8 | Sarah    | Parker   | Teller         |
|      9 | Jane     | Grossman | Teller         |
|     11 | Thomas   | Ziegler  | Teller         |
|     12 | Samantha | Jameson  | Teller         |
|     14 | Cindy    | Mason    | Teller         |
|     15 | Frank    | Portman  | Teller         |
|     17 | Beth     | Fowler   | Teller         |
|     18 | Rick     | Tulman   | Teller         |
+--------+----------+----------+----------------+
11 rows in set (0.00 sec)
```

这个查询检索所有不管理别人的雇员。在这个查询中，我为子查询添加了一个过滤条件以确保 null 值不会出现在子查询的返回表中（后面将会解释为什么在这种情况下需要如此过滤）。

all 运算符

in 运算符被用于查看是否能在一个表达式集合中找到某一个表达式，all 运算符则用于将某单值与集合中的每个值进行比较。构建这样的条件需要将其中一个比较运算符（=、<>、<、> 等）与 all 运算符配合使用。例如，下面的查询就是查找雇员 ID 与任何主管 ID 不同的所有雇员。

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id <> ALL (SELECT superior_emp_id
    ->   FROM employee
    ->   WHERE superior_emp_id IS NOT NULL);
+--------+----------+----------+----------------+
| emp_id | fname    | lname    | title          |
+--------+----------+----------+----------------+
|      2 | Susan    | Barker   | Vice President |
|      5 | John     | Gooding  | Loan Manager   |
|      7 | Chris    | Tucker   | Teller         |
|      8 | Sarah    | Parker   | Teller         |
|      9 | Jane     | Grossman | Teller         |
|     11 | Thomas   | Ziegler  | Teller         |
|     12 | Samantha | Jameson  | Teller         |
|     14 | Cindy    | Mason    | Teller         |
|     15 | Frank    | Portman  | Teller         |
|     17 | Beth     | Fowler   | Teller         |
|     18 | Rick     | Tulman   | Teller         |
+--------+----------+----------+----------------+
11 rows in set (0.05 sec)
```

这个子查询再次返回所有主管的 ID，同时包含查询返回 ID 不等于子查询结果集中所有 ID 的每个雇员数据。换句话说，查询就是检索非主管雇员。读者如果感到这种方法有些笨拙，请不必为此担忧，大多数人也是如此想的，他们宁愿构造不同的查询短语来避免使用 all 运算符。例如，这个查询和上一个例子的结果相同，后者却使用了 not in 运算符。事实上，采用哪种方法只是一个喜好问题，不过我以为大多数人都认为使用 not in 要更容易理解。

提示

当使用 not in 或 <>运算符比较一个值和一个值集时，读者必须注意确保值集中不包含 null 值，这是因为服务器将表达式左边的值与值集中的每个成员比较时，可能出现该值与 null 比较的情况，而任何一个将值与 null 进行比较的企图都将产生未知的结果。因此，下面的查询返回一个空结果集。

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id NOT IN (1, 2, NULL);
Empty set (0.00 sec)
```

在某些情况下，all 运算符比其他运算符更适于使用。下面的例子是用 all 查找可用余额小于 Frank Tucker 所有账户的账户。

```
mysql> SELECT account_id, cust_id, product_cd, avail_balance
    -> FROM account
    -> WHERE avail_balance < ALL (SELECT a.avail_balance
    ->   FROM account a INNER JOIN individual i
    ->     ON a.cust_id = i.cust_id
    ->   WHERE i.fname = 'Frank' AND i.lname = 'Tucker');
+------------+---------+------------+---------------+
|account_id  | cust_id | product_cd |avail_balance  |
+------------+---------+------------+---------------+
|          2 |       1 | SAV        |        500.00 |
|          5 |       2 | SAV        |        200.00 |
|         10 |       4 | CHK        |        534.12 |
|         11 |       4 | SAV        |        767.77 |
|         14 |       6 | CHK        |        122.37 |
|         19 |       8 | SAV        |        387.99 |
|         21 |       9 | CHK        |        125.67 |
|         25 |      10 | BUS        |          0.00 |
+------------+---------+------------+---------------+
8 rows in set (0.17 sec)
```

下面的数据是子查询返回的 Frank 每个账户的可用余额：

```
mysql> SELECT a.avail_balance
    -> FROM account a INNER JOIN individual i
    ->   ON a.cust_id = i.cust_id
    -> WHERE i.fname = 'Frank' AND i.lname = 'Tucker';
+---------------+
| avail_balance |
+---------------+
|       1057.75 |
|       2212.50 |
+---------------+
2 rows in set (0.01 sec)
```

Frank 有两个账户，其中最小的余额是$1057.75。包含查询检索余额小于 Frank 任一账户的所有账户，因此结果集包含余额小于$1057.75 的所有账户。

any 运算符

与 all 运算符一样，any 运算符允许将一个值与值集中每个成员相比较。与 all 不同的是，使用 any 运算符时，只要有一个比较成立，则条件为真；使用 all 运算符时，只有与集合中的所有成员比较都成立时条件才为真。例如，读者可以如下查找可用余额大于 Frank Tucker 任意账户的所有账户：

```
mysql> SELECT account_id, cust_id, product_cd, avail_balance
    -> FROM account
```

```
    -> WHERE avail_balance > ANY (SELECT a.avail_balance
    ->   FROM account a INNER JOIN individual i
    ->     ON a.cust_id = i.cust_id
    ->   WHERE i.fname = 'Frank' AND i.lname = 'Tucker');
+------------+---------+------------+---------------+
| account_id | cust_id | product_cd | avail_balance |
+------------+---------+------------+---------------+
|          3 |       1 | CD         |       3000.00 |
|          4 |       2 | CHK        |       2258.02 |
|          8 |       3 | MM         |       2212.50 |
|         12 |       4 | MM         |       5487.09 |
|         13 |       5 | CHK        |       2237.97 |
|         15 |       6 | CD         |      10000.00 |
|         17 |       7 | CD         |       5000.00 |
|         18 |       8 | CHK        |       3487.19 |
|         22 |       9 | MM         |       9345.55 |
|         23 |       9 | CD         |       1500.00 |
|         24 |      10 | CHK        |      23575.12 |
|         27 |      11 | BUS        |       9345.55 |
|         28 |      12 | CHK        |      38552.05 |
|         29 |      13 | SBL        |      50000.00 |
+------------+---------+------------+---------------+
14 rows in set (0.00 sec)
```

Frank 有两个余额分别为$1 057.75 和 $2 212.50 的账户，那么账户的余额最少为$1 057.75 时才能称作它的余额大于这两个账户中的任意一个。

提示

大多数人喜欢使用 in 运算符，不过使用= any 与使用 in 等效。

9.3.2 多列子查询

直到现在，本章的所有子查询例子都是返回单列单行或者多行结果。不过，在某些情况下，读者可以使用返回两列或者多列的子查询。为了展示多列子查询的用途，下面先看看一个多重单列子查询的例子：

```
mysql> SELECT account_id, product_cd, cust_id
    -> FROM account
    -> WHERE open_branch_id = (SELECT branch_id
    ->   FROM branch
    ->   WHERE name = 'Woburn Branch')
    ->   AND open_emp_id IN (SELECT emp_id
    ->   FROM employee
    ->   WHERE title = 'Teller' OR title = 'Head Teller');
+------------+------------+---------+
| account_id | product_cd | cust_id |
+------------+------------+---------+
```

```
|          1 | CHK        |        1 |
|          2 | SAV        |        1 |
|          3 | CD         |        1 |
|          4 | CHK        |        2 |
|          5 | SAV        |        2 |
|         17 | CD         |        7 |
|         27 | BUS        |       11 |
+------------+------------+---------+
7 rows in set (0.09 sec)
```

这个查询使用两个子查询检索出 Woburn 分行的 ID 以及所有银行柜台员的 ID，同时包含查询使用这个信息查找所有 Woburn 分行柜台员开立的账户。不过，既然 employee 表包含每个柜台员所属分行的信息，那么将 account.open_branch_id 和 account.open_emp_id 两列与对 employee 和 branch 两个表的单一子查询结果作比较也可以获得同样的结果。因此，过滤条件必须将这两列用括号括起来，并且排列顺序与子查询结果的顺序相同。

```
mysql> SELECT account_id, product_cd, cust_id
    -> FROM account
    -> WHERE (open_branch_id, open_emp_id) IN
    ->  (SELECT b.branch_id, e.emp_id
    ->   FROM branch b INNER JOIN employee e
    ->     ON b.branch_id = e.assigned_branch_id
    ->   WHERE b.name = 'Woburn Branch'
    ->     AND (e.title = 'Teller' OR e.title = 'Head Teller'));
+------------+------------+---------+
| account_id | product_cd | cust_id |
+------------+------------+---------+
|          1 | CHK        |       1 |
|          2 | SAV        |       1 |
|          3 | CD         |       1 |
|          4 | CHK        |       2 |
|          5 | SAV        |       2 |
|         17 | CD         |       7 |
|         27 | BUS        |      11 |
+------------+------------+---------+
7 rows in set (0.00 sec)
```

这种方式的查询功能与前面的例子相同，只不过是使用返回两列的单一查询代替两个只返回单列的子查询。

当然，读者可以简单地连接 3 个表代替子查询来重写上面的例子，但是学会通过多种方法获得同样的结果对于学习 SQL 是非常有帮助的。下面的例子必须要有子查询。假设一些客户投诉 account 表中的可用余额和待收余额不正确，下面是查找余额与交易金额总量不匹配的账户的部分解决方案：

```
SELECT 'ALERT! : Account #1 Has Incorrect Balance!'
FROM account
```

```
WHERE (avail_balance, pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
   SUM(<expression to generate pending balance>)
  FROM transaction
  WHERE account_id = 1)
  AND account_id = 1;
```

正如读者所见，我并没有完成计算交易总额的表达式，从而计算可用余额和待收余额。在第 11 章中探讨如何构造 case 表达式后，我将会完成这个工作。不过，从这个查询已经可以明白子查询是先从 transaction 表获取的两个总和，然后将其与 account 表中的 avail_balance 和 pending_balance 列的值进行比较。子查询和包含查询中都添加了过滤条件 account_id = 1，这样当前的查询一次只会检索一个账户。在下一节中，读者将学习如何写一个通式查询，它可以一次执行而查询所有账户。

9.4 关联子查询

一方面，到目前为止范例中的所有子查询都是独立于包含语句的，这意味着这些子查询可以被单独执行，并可检验结果；另一方面，关联子查询依附于包含语句并引用其一列或者多列。与非关联子查询不同，关联子查询不是在包含语句执行之前一次执行完毕，而是为每一个候选行（这些行可能会包含在最终结果里）执行一次。例如，下面的查询首先利用关联查询计算每个客户的账户数，接着包含查询检索出那些拥有两个账户的客户：

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city
    -> FROM customer c
    -> WHERE 2 = (SELECT COUNT(*)
    ->   FROM account a
    ->   WHERE a.cust_id = c.cust_id);
+---------+--------------+---------+
| cust_id | cust_type_cd | city    |
+---------+--------------+---------+
|       2 | I            | Woburn  |
|       3 | I            | Quincy  |
|       6 | I            | Waltham |
|       8 | I            | Salem   |
|      10 | B            | Salem   |
+---------+--------------+---------+
5 rows in set (0.01 sec)
```

子查询最后引用的 c.cust_id 使之具有关联性，这样它的执行必须依赖于包含查询提供的 c.cust_id。在这种情况下，先从 customer 表中检索出 13 行客户记录，接着为每个客户执行一次子查询，每次执行包含查询都要向子查询传递客户 ID。若子查询返回值 2，则过滤条件满足，该行将被添加到结果集。

除了等式条件，关联子查询还可以用于其他类型的条件，比如下面的范围条件：

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city
    -> FROM customer c
    -> WHERE (SELECT SUM(a.avail_balance)
    ->    FROM account a
    ->    WHERE a.cust_id = c.cust_id)
    ->  BETWEEN 5000 AND 10000;
+---------+-------------+------------+
| cust_id | cust_type_cd | city       |
+---------+-------------+------------+
|       4 | I           | Waltham    |
|       7 | I           | Wilmington |
|      11 | B           | Wilmington |
+---------+-------------+------------+
3 rows in set (0.02 sec)
```

本查询与前一个查询的不同之处在于它检索的是所有账户余额总和在\$5 000 和\$10 000 之间的所有客户。同样，关联子查询也执行 13 次（为每个客户执行一次），同时每次执行返回指定客户的所有账户余额总和。

提示

与前面查询的另一个微妙的区别是子查询在条件的左边。这看上去有点奇怪，但确实是有效的。

在前一节的最后，我说明了如何通过账户交易日志查询账户可用余额和待收余额，并且约定要向读者展示如何修改例子已达到一次执行而处理所有账户。下面再看看这个例子：

```
SELECT 'ALERT! : Account #1 Has Incorrect Balance!'
FROM account
WHERE (avail_balance, pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
   SUM(<expression to generate pending balance>)
  FROM transaction
  WHERE account_id = 1)
  AND account_id = 1;
```

如果用关联子查询替代非关联子查询，那么包含查询执行一次，而子查询要为每个账户运行一次。下面是前面查询的升级版本：

```
SELECT CONCAT('ALERT! : Account #', a.account_id,
  'Has Incorrect Balance!')
FROM account a
WHERE (a.avail_balance, a.pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
   SUM(<expression to generate pending balance>)
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

现在的子查询包括一个过滤条件，它将交易账户 ID 与从包含查询引用的账户 ID 连接。select 子句的警示信息也被修改成连接一个包含账户的 ID，而不只是一个硬编码值 1。

9.4.1 exists 运算符

读者将会经常看到关联子查询应用在等式和范围条件中，但实际上 exists 运算符是构造包含关联子查询条件的最常用运算符。若只关心存在关系而不在乎数量，就可以使用 exists 运算符。例如，下面的查询就是检索在特定日期进行过交易的所有账户，并不关心到底进行了多少次交易：

```
SELECT a.account_id, a.product_cd, a.cust_id, a.avail_balance
FROM account a
WHERE EXISTS (SELECT 1
  FROM transaction t
  WHERE t.account_id = a.account_id
    AND t.txn_date = '2008-09-22');
```

使用 exists 运算符时，子查询可能会返回 0、1 或者多行结果，然而条件只是简单地检查子查询能否返回至少 1 行。读者看看子查询中的 select 子句就会发现它只由单个文字 1 组成，这是因为包含查询的条件只需要知道子查询返回的结果是多少行，而与结果的确切内容无关。事实上，读者可以让子查询返回任何自己喜欢的结果，例如：

```
SELECT a.account_id, a.product_cd, a.cust_id, a.avail_balance
FROM account a
WHERE EXISTS (SELECT t.txn_id, 'hello', 3.1415927
  FROM transaction t
  WHERE t.account_id = a.account_id
    AND t.txn_date = '2008-09-22');
```

不过，惯例是 select 1 或者 select * 。

读者也可以使用 not exists 运算符检查子查询返回行数是否为 0，具体如下：

```
mysql> SELECT a.account_id, a.product_cd, a.cust_id
    -> FROM account a
    -> WHERE NOT EXISTS (SELECT 1
    ->   FROM business b
    ->   WHERE b.cust_id = a.cust_id);
+------------+------------+---------+
| account_id | product_cd | cust_id |
+------------+------------+---------+
|          1 | CHK        |       1 |
|          2 | SAV        |       1 |
|          3 | CD         |       1 |
|          4 | CHK        |       2 |
|          5 | SAV        |       2 |
|          7 | CHK        |       3 |
|          8 | MM         |       3 |
|         10 | CHK        |       4 |
```

```
|          11 | SAV         |       4 |
|          12 | MM          |       4 |
|          13 | CHK         |       5 |
|          14 | CHK         |       6 |
|          15 | CD          |       6 |
|          17 | CD          |       7 |
|          18 | CHK         |       8 |
|          19 | SAV         |       8 |
|          21 | CHK         |       9 |
|          22 | MM          |       9 |
|          23 | CD          |       9 |
+------------+-----------+---------+
19 rows in set (0.99 sec)
```

上面的查询检索客户 ID 没有出现在 business 表中的所有客户，这也是一种查找所有非商业客户的间接方法。

9.4.2 关联子查询操作数据

到目前为止，本章的所有例子都是 select 语句，但这并不意味着子查询在其他 SQL 语句中没有用处。子查询也大量应用于 update、delete 和 insert 语句，并且关联子查询也会频繁出现于 update 和 delete 语句中。下面的关联子查询用于修改 account 表中的 last_activity_date 列：

```
UPDATE account a
SET a.last_activity_date =
 (SELECT MAX(t.txn_date)
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

上面的语句查找每个账户的最新交易日期，再修改 account 表中每一行（因为没有 where 子句）的 last_activity_date 列的值。虽然期望每个账户至少存在一个交易与其相连接似乎是合理的，但最好还是在打算修改 last_activity_date 列之前检查每个账户是否发生过交易；否则，由于子查询不返回任何行，last_activity_date 列的值将被修改为 null。下面是 update 语句的另一个版本，不过这一次增加了一个具有关联子查询的 where 子句：

```
UPDATE account a
SET a.last_activity_date =
 (SELECT MAX(t.txn_date)
  FROM transaction t
  WHERE t.account_id = a.account_id)
WHERE EXISTS (SELECT 1
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

除了 select 子句外，这两个关联子查询相同。不过，set 子句的子查询仅当 update 语句中 where 子句为真时才执行（这意味着至少能查找到一次这个账户的交易记录），这样

就可以保护 last_activity_date 列不被 null 重写。

关联子查询也经常应用于 delete 语句。例如，读者可以在周末运行数据维护脚本，删除不需要的数据。这个脚本可能包含了下面的语句，它从 department 表删除没有子行出现在 employee 表中的所有行：

```
DELETE FROM department
WHERE NOT EXISTS (SELECT 1
  FROM employee
  WHERE employee.dept_id = department.dept_id);
```

切记，在 MySQL 中 delete 语句使用关联子查询时，无论如何都不能使用表别名，这就是为什么我在子查询中使用表全名的原因。不过，在多数其他数据库服务器中，department 表和 employee 表是可以使用别名的，例如：

```
DELETE FROM department d
WHERE NOT EXISTS (SELECT 1
  FROM employee e
  WHERE e.dept_id = d.dept_id);
```

9.5　何时使用子查询

读者已经学习了不同类型的子查询和用于实现与子查询返回的数据交互的各种运算符，现在开始探讨用子查询构建强大的 SQL 语句的各种方法。接下来将向读者说明如何使用子查询创建自定义表、构造条件以及在结果集中生成列的值。

9.5.1　子查询作为数据源

早在第 3 章中我就说明了 select 语句中 from 子句的作用是列举需要查询的表。子查询生成的结果集包含行、列数据，因而非常适合将它与表一起包含在 from 子句的子查询里。乍一看，子查询与表联合使用似乎只是一个有趣而毫无意义的特性，但事实上这是编写查询时可用的最强大工具之一。下面是一个简单的例子：

```
mysql> SELECT d.dept_id, d.name, e_cnt.how_many num_employees
    -> FROM department d INNER JOIN
    ->  (SELECT dept_id, COUNT(*) how_many
    ->   FROM employee
    ->   GROUP BY dept_id) e_cnt
    ->   ON d.dept_id = e_cnt.dept_id;
+---------+----------------+---------------+
| dept_id | name           | num_employees |
+---------+----------------+---------------+
|       1 | Operations     |            14 |
|       2 | Loans          |             1 |
|       3 | Administration |             3 |
```

```
+---------+--------------+--------------+
3 rows in set (0.04 sec)
```

在上面的例子中，子查询生成了部门 ID 及其对应的雇员数。下面就是子查询生成的结果集：

```
mysql> SELECT dept_id, COUNT(*) how_many
    -> FROM employee
    -> GROUP BY dept_id;
+---------+----------+
| dept_id | how_many |
+---------+----------+
|       1 |       14 |
|       2 |        1 |
|       3 |        3 |
+---------+----------+
3 rows in set (0.00 sec)
```

子查询被命名为 e_cnt，并且用 dept_id 列 e_cnt 与 department 表连接，然后包含查询从 department 表检索出部门 ID、名称以及来自 e_cnt 子查询的雇员数。

子查询在 from 子句中使用必须是非关联的，它们首先执行，然后一直保留于内存中直至包含查询执行完毕。子查询为写查询语句提供了极大的可扩展性，因为它使读者可以远远超越可用基础表集合，几乎能够创建自己需要的任何数据视图，进而将这些结果与表或者其他子查询连接。因而，如果需要创建报表或者是为外部系统生成数据源，读者就能够用单一查询解决了，过去则需要多重查询或者过程语言来完成。

数据加工

除了使用查询总结现有数据，读者还可以生成数据库中不存在的数据。例如，打算按照储蓄账户里余额的多少对客户分组，但是这些组的定义根本没有存储在数据库中。例中，现在需要将客户按照表 9-1 的组定义进行分类。

表 9-1　客户余额分组

组　　名	下　限　值	上　限　值
Small Fry	0	$4999.99
Average Joes	$5000	$9999.99
Heavy Hitters	$10000	$9999999999

为了在单一查询里生成这些组，读者需要一种方法来定义这些组。第一步是定义一个用于生成组定义的查询：

```
mysql> SELECT 'Small Fry' name, 0 low_limit, 4999.99 high_limit
    -> UNION ALL
    -> SELECT 'Average Joes' name, 5000 low_limit, 9999.99 high_limit
    -> UNION ALL
```

```
    -> SELECT 'Heavy Hitters' name, 10000 low_limit, 9999999.99
high_limit;
+---------------+-----------+------------+
| name          | low_limit | high_limit |
+---------------+-----------+------------+
| Small Fry     |         0 |    4999.99 |
| Average Joes  |      5000 |    9999.99 |
| Heavy Hitters |     10000 | 9999999.99 |
+---------------+-----------+------------+
3 rows in set (0.00 sec)
```

我在上面的语句中使用了集合运算符 union all 将来自 3 个独立查询的结果合并成一个单一的结果集。每个查询返回 3 个文字，然后将这些结果再组合，从而生成 3 行 3 列的结果集。现在已经有了生成组的查询，只要将它添加到另一个查询的 from 子句中就可以了：

```
mysql> SELECT groups.name, COUNT(*) num_customers
    -> FROM
    ->  (SELECT SUM(a.avail_balance) cust_balance
    ->   FROM account a INNER JOIN product p
    ->     ON a.product_cd = p.product_cd
    ->   WHERE p.product_type_cd = 'ACCOUNT'
    ->   GROUP BY a.cust_id) cust_rollup
    ->   INNER JOIN
    ->  (SELECT 'Small Fry' name, 0 low_limit, 4999.99 high_limit
    ->   UNION ALL
    ->   SELECT 'Average Joes' name, 5000 low_limit,
    ->     9999.99 high_limit
    ->   UNION ALL
    ->   SELECT 'Heavy Hitters' name, 10000 low_limit,
    ->     9999999.99 high_limit) groups
    ->   ON cust_rollup.cust_balance
    ->     BETWEEN groups.low_limit AND groups.high_limit
    -> GROUP BY groups.name;
+---------------+---------------+
| name          | num_customers |
+---------------+---------------+
| Average Joes  |             2 |
| Heavy Hitters |             4 |
| Small Fry     |             5 |
+---------------+---------------+
3 rows in set (0.01 sec)
```

上面包含查询的 from 子句有两个子查询：第一个名为 cust_rollup，返回每个客户的储蓄余额总和；另一个名为 groups，生成 3 个客户分组。cust_rollup 生成的数据如下：

```
mysql> SELECT SUM(a.avail_balance) cust_balance
    -> FROM account a INNER JOIN product p
    ->   ON a.product_cd = p.product_cd
```

```
    -> WHERE p.product_type_cd = 'ACCOUNT'
    -> GROUP BY a.cust_id;
+--------------+
| cust_balance |
+--------------+
|      4557.75 |
|      2458.02 |
|      3270.25 |
|      6788.98 |
|      2237.97 |
|     10122.37 |
|      5000.00 |
|      3875.18 |
|     10971.22 |
|     23575.12 |
|     38552.05 |
+--------------+
11 rows in set (0.05 sec)
```

接着，cust_rollup 生成的数据通过范围条件（cust_rollup.cust_balance BETWEEN groups.low_limit AND groups.high_limit）与 groups 表连接。最后，连接的数据被分组并计算每个组的客户数，进而产生最后的结果集。

当然，读者也可以不使用子查询，而是简单地创建一个固定的表来描述组的定义。不过，读者在使用这种方法后，不久可能会发现数据库因为这些用于特殊目的的表而变得杂乱，并且忘记了大多数表是为了什么而创建的。我曾经工作过的一个环境允许数据库的使用者为特殊目的创建他们自己的表，这样就造成了很多灾难性的后果（表没有备份、数据库升级时表丢失以及由于空间分配问题而导致的服务器停机等）。然而，有了子查询的帮助后，读者应该坚持这样一个原则：仅当有明确的商业需求保存这些新数据时才能添加相应的新表到数据库。

面向任务的子查询

在用于生成报告或数据源的系统里，读者经常会遇到如下查询：

```
mysql> SELECT p.name product, b.name branch,
    ->  CONCAT(e.fname, ' ', e.lname) name,
    ->  SUM(a.avail_balance) tot_deposits
    -> FROM account a INNER JOIN employee e
    ->  ON a.open_emp_id = e.emp_id
    ->  INNER JOIN branch b
    ->  ON a.open_branch_id = b.branch_id
    ->  INNER JOIN product p
    ->  ON a.product_cd = p.product_cd
    -> WHERE p.product_type_cd = 'ACCOUNT'
    -> GROUP BY p.name, b.name, e.fname, e.lname
    -> ORDER BY 1,2;
```

```
+--------------------+-------------+--------------+--------------+
| product            | branch      | name         | tot_deposits |
+--------------------+-------------+--------------+--------------+
| certificate of deposit| Headquarters | Michael Smith |  11500.00  |
| certificate of deposit| Woburn Branch | Paula Roberts |   8000.00  |
| checking account   | Headquarters | Michael Smith |    782.16  |
| checking account   | Quincy Branch | John Blake    |   1057.75  |
| checking account   | So. NH Branch | Theresa Markham| 67852.33  |
| checking account   | Woburn Branch | Paula Roberts |   3315.77  |
| money market account | Headquarters | Michael Smith |  14832.64  |
| money market account | Quincy Branch | John Blake    |   2212.50  |
| savings account    | Headquarters | Michael Smith |    767.77  |
| savings account    | So. NH Branch | Theresa Markham|   387.99  |
| savings account    | Woburn Branch | Paula Roberts |    700.00  |
+--------------------+-------------+--------------+--------------+
11 rows in set (0.00 sec)
```

上面的查询依据账户类型、开户雇员以及开户行对所有储蓄账户余额求和。读者如果仔细看这个查询就会发现 product、branch 和 employee 这 3 个表只是用于描述目的，并且 account 表已经有了生成分组（product_cd、open_branch_id、open_emp_id 和 avail_balance）所需的一切。因此，读者可以将生成分组的任务独立出来，生成一个子查询，然后将子查询生成的 3 个表连接成一个表，最后得到最终结果。下面是分组的子查询：

```
mysql> SELECT product_cd, open_branch_id branch_id, open_emp_id emp_id,
    ->   SUM(avail_balance) tot_deposits
    -> FROM account
    -> GROUP BY product_cd, open_branch_id, open_emp_id;
+------------+-----------+--------+--------------+
| product_cd | branch_id | emp_id | tot_deposits |
+------------+-----------+--------+--------------+
| BUS        |         2 |     10 |      9345.55 |
| BUS        |         4 |     16 |         0.00 |
| CD         |         1 |      1 |     11500.00 |
| CD         |         2 |     10 |      8000.00 |
| CHK        |         1 |      1 |       782.16 |
| CHK        |         2 |     10 |      3315.77 |
| CHK        |         3 |     13 |      1057.75 |
| CHK        |         4 |     16 |     67852.33 |
| MM         |         1 |      1 |     14832.64 |
| MM         |         3 |     13 |      2212.50 |
| SAV        |         1 |      1 |       767.77 |
| SAV        |         2 |     10 |       700.00 |
| SAV        |         4 |     16 |       387.99 |
| SBL        |         3 |     13 |     50000.00 |
+------------+--------- -+--------+--------------+
14 rows in set (0.02 sec)
```

上面的查询正是这个查询的核心，需要的其他表只是将 product_cd、 open_branch_id 和 open_emp_id 这些外键替换为有意义的字符串。下面的查询将对 account 表的查询包装在一个子查询中，并将其结果表与其他 3 个表连接。

```
mysql> SELECT p.name product, b.name branch,
    ->  CONCAT(e.fname, ' ', e.lname) name,
    ->  account_groups.tot_deposits
    -> FROM
    -> (SELECT product_cd, open_branch_id branch_id,
    ->    open_emp_id emp_id,
    ->    SUM(avail_balance) tot_deposits
    ->  FROM account
    ->  GROUP BY product_cd, open_branch_id, open_emp_id) account_groups
    ->  INNER JOIN employee e ON e.emp_id = account_groups.emp_id
    ->  INNER JOIN branch b ON b.branch_id = account_groups.branch_id
    ->  INNER JOIN product p ON p.product_cd = account_groups.product_cd
    -> WHERE p.product_type_cd = 'ACCOUNT';
+--------------------+-------------+---------------+-------------+
| product            | branch      | name          | tot_deposits |
+--------------------+-------------+---------------+-------------+
| certificate of deposit | Headquarters  | Michael Smith   |  11500.00    |
| certificate of deposit | Woburn Branch | Paula Roberts   |   8000.00    |
| checking account   | Headquarters  | Michael Smith   |    782.16    |
| checking account   | Quincy Branch | John Blake      |   1057.75    |
| checking account   | So. NH Branch | Theresa Markham | 67852.33     |
| checking account   | Woburn Branch | Paula Roberts   |   3315.77    |
| money market account | Headquarters | Michael Smith  |  14832.64    |
| money market account | Quincy Branch | John Blake    |   2212.50    |
| savings account    | Headquarters  | Michael Smith   |    767.77    |
| savings account    | So. NH Branch | Theresa Markham |    387.99    |
| savings account    | Woburn Branch | Paula Roberts   |    700.00    |
+--------------------+-------------+---------------+-------------+
11 rows in set (0.01 sec)
```

常言道，情人眼里出西施，就这两个版本而言，个人喜好不同，不过我还是以为这个版本比那个大而扁平的版本更加令人满意。这个版本的查询执行也更快，因为分组的实现不是基于可能很长的字符串型列（branch.name、product.name、employee.fname、employee.lname），而是基于更小而数字型的外键列（product_cd、open_branch_id、open_emp_id）。

9.5.2 过滤条件中的子查询

本章许多例子都是把子查询用作过滤条件的表达式，事实上，这就是子查询的主要用途之一。不过，过滤条件中的子查询并不会只出现在 where 子句中。例如，下面的查询就是在 having 子句中使用子查询来查找开户最多的雇员：

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
```

```
    -> GROUP BY open_emp_id
    -> HAVING COUNT(*) = (SELECT MAX(emp_cnt.how_many)
    ->   FROM (SELECT COUNT(*) how_many
    ->     FROM account
    ->     GROUP BY open_emp_id) emp_cnt);
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
+-------------+----------+
1 row in set (0.01 sec)
```

having 子句中的子查询检索所有雇员的最大开户数，包含查询则查找这个开户最多的雇员。如果有多人并列开户数最高，那么查询将返回多行。

9.5.3 子查询作为表达式生成器

作为本节的最后一小节，我们将继续完成开始学的内容：单列单行的标量子查询。除了用于过滤条件中，标量子查询还能用在表达式可以出现的任何位置，其中包括查询中的 select 和 order by 子句以及 insert 语句中的 values 子句。

在“面向任务的子查询”中，我展示了如何将分组从其他查询中分离出来。下面的查询语句是前面查询的另一个版本，采用子查询实现了相同的目的，但方法不同：

```
mysql> SELECT
    ->  (SELECT p.name FROM product p
    ->   WHERE p.product_cd = a.product_cd
    ->     AND p.product_type_cd = 'ACCOUNT') product,
    ->  (SELECT b.name FROM branch b
    ->   WHERE b.branch_id = a.open_branch_id) branch,
    ->  (SELECT CONCAT(e.fname, ' ', e.lname) FROM employee e
    ->   WHERE e.emp_id = a.open_emp_id) name,
    ->   SUM(a.avail_balance) tot_deposits
    -> FROM account a
    -> GROUP BY a.product_cd, a.open_branch_id, a.open_emp_id
    -> ORDER BY 1,2;
+--------------------+-------------+-------------+-------------+
| product            | branch      | name        | tot_deposits |
+--------------------+-------------+-------------+-------------+
| NULL               | Quincy Branch | John Blake      |   50000.00    |
| NULL               | So. NH Branch | Theresa Markham |      0.00     |
| NULL               | Woburn Branch | Paula Roberts   |    9345.55    |
| certificate of deposit | Headquarters  | Michael Smith   |   11500.00    |
| certificate of deposit | Woburn Branch | Paula Roberts   |    8000.00    |
| checking account   | Headquarters  | Michael Smith   |     782.16    |
| checking account   | Quincy Branch | John Blake      |    1057.75    |
| checking account   | So. NH Branch | Theresa Markham |   67852.33    |
| checking account   | Woburn Branch | Paula Roberts   |    3315.77    |
| money market account | Headquarters  | Michael Smith   |   14832.64    |
```

```
| money market account | Quincy Branch | John Blake      |   2212.50     |
| savings account      | Headquarters  | Michael Smith   |    767.77     |
| savings account      | So. NH Branch | Theresa Markham |    387.99     |
| savings account      | Woburn Branch | Paula Roberts   |    700.00     |
+--------------------+-------------+-------------+------------+
14 rows in set (0.01 sec)
```

这个查询与前面介绍的最大差别在于它在 select 子句中使用了子查询：

- 不是将 product、branch 和 employee 与账户数据连接，而是在 select 子句中使用关联标量子查询查找产品、分行和雇员的名字；
- 结果有 14 行，而不是 11 行，其中 3 个产品名称为 null。

结果集中有 3 行比较特殊的原因是前面的版本中有一个过滤条件 p.product_type_cd='ACCOUNT'，而这个条件排除了 INSURANCE 和 LOAN 类型的产品，比如小额商业贷款。由于这个版本的查询没有连接到 product 表，因此就没有办法在主查询里添加这个过滤条件。没有包括这个条件对于 product 表的关联子查询的唯一影响就是留下了 null 这样的产品名。因此，读者如果想去掉这 3 个特殊行，就需要将 product 表与 account 表连接，同时添加这个过滤条件，或者干脆作如下简单修改：

```
mysql> SELECT all_prods.product, all_prods.branch,
    ->  all_prods.name, all_prods.tot_deposits
    -> FROM
    ->  (SELECT
    ->    (SELECT p.name FROM product p
    ->     WHERE p.product_cd = a.product_cd
    ->       AND p.product_type_cd = 'ACCOUNT') product,
    ->    (SELECT b.name FROM branch b
    ->     WHERE b.branch_id = a.open_branch_id) branch,
    ->    (SELECT CONCAT(e.fname, ' ', e.lname) FROM employee e
    ->     WHERE e.emp_id = a.open_emp_id) name,
    ->     SUM(a.avail_balance) tot_deposits
    ->   FROM account a
    ->   GROUP BY a.product_cd, a.open_branch_id, a.open_emp_id
    ->  ) all_prods
    -> WHERE all_prods.product IS NOT NULL
    -> ORDER BY 1,2;
+------------------+-------------+--------------+-------------+
| product                | branch        | name            | tot_deposits  |
+------------------+-------------+--------------+-------------+
| certificate of deposit | Headquarters  | Michael Smith   |  11500.00     |
| certificate of deposit | Woburn Branch | Paula Roberts   |   8000.00     |
| checking account       | Headquarters  | Michael Smith   |    782.16     |
| checking account       | Quincy Branch | John Blake      |   1057.75     |
| checking account       | So. NH Branch | Theresa Markham |  67852.33     |
| checking account       | Woburn Branch | Paula Roberts   |   3315.77     |
| money market account   | Headquarters  | Michael Smith   |  14832.64     |
| money market account   | Quincy Branch | John Blake      |   2212.50     |
```

```
| savings account     | Headquarters   | Michael Smith   |    767.77      |
| savings account     | So. NH Branch  | Theresa Markham |    387.99      |
| savings account     | Woburn Branch  | Paula Roberts   |    700.00      |
+--------------------+---------------+---------------+--------------+
11 rows in set (0.01 sec)
```

将前面的查询简单地包装到一个子查询（all_prods）里，同时添加过滤条件，排除那些 product 列的值为 null 值的行， 这样查询就只返回所需的 11 行了。查询的最终结果是对 account 表中的原始数据分类，而其他 3 个表中的数据是用于润色输出的数据。

前面已经提到，标量子查询也可以出现在 order by 子句里。下面的查询检索雇员数据，其结果排序的第一准则是雇员老板的姓氏，第二准则是雇员的姓氏：

```
mysql> SELECT emp.emp_id, CONCAT(emp.fname, ' ', emp.lname) emp_name,
    ->  (SELECT CONCAT(boss.fname, ' ', boss.lname)
    ->   FROM employee boss
    ->   WHERE boss.emp_id = emp.superior_emp_id) boss_name
    -> FROM employee emp
    -> WHERE emp.superior_emp_id IS NOT NULL
    -> ORDER BY (SELECT boss.lname FROM employee boss
    ->   WHERE boss.emp_id = emp.superior_emp_id), emp.lname;
+--------+------------------+------------------+
| emp_id | emp_name         | boss_name        |
+--------+------------------+------------------+
|     14 | Cindy Mason      | John Blake       |
|     15 | Frank Portman    | John Blake       |
|      9 | Jane Grossman    | Helen Fleming    |
|      8 | Sarah Parker     | Helen Fleming    |
|      7 | Chris Tucker     | Helen Fleming    |
|     13 | John Blake       | Susan Hawthorne  |
|      6 | Helen Fleming    | Susan Hawthorne  |
|      5 | John Gooding     | Susan Hawthorne  |
|     16 | Theresa Markham  | Susan Hawthorne  |
|     10 | Paula Roberts    | Susan Hawthorne  |
|     17 | Beth Fowler      | Theresa Markham  |
|     18 | Rick Tulman      | Theresa Markham  |
|     12 | Samantha Jameson | Paula Roberts    |
|     11 | Thomas Ziegler   | Paula Roberts    |
|      2 | Susan Barker     | Michael Smith    |
|      3 | Robert Tyler     | Michael Smith    |
|      4 | Susan Hawthorne  | Robert Tyler     |
+--------+------------------+------------------+
17 rows in set (0.01 sec)
```

这个查询使用了两个关联标量子查询：一个是检索每个雇员老板姓名的 select 子句，另一个是用于排序而只返回每个雇员老板姓氏的 order by 子句。

除了在 select 子句中使用关联标量子查询，读者还可以用非关联标量子查询为 insert 子句生成值。比如说要生成一个新的账户，相关的数据如下：

- 产品名称（“savings account”）；
- 客户联邦个人识别号码（“555-55-5555”）；
- 开户行名称（“Quincy Branch”）；
- 开户柜员的姓名。

为了能够填充 account 表中的外键列，读者在创建新行之前应该查找所有数据的键值。那么应该如何做呢？有两个选择：一个是先用 4 个查询检索到主关键字，再将其置于 insert 语句里；另一个是直接用子查询在 insert 语句里检索这 4 个键值：

```
INSERT INTO account
 (account_id, product_cd, cust_id, open_date, last_activity_date,
  status, open_branch_id, open_emp_id, avail_balance, pending_balance)
VALUES (NULL,
 (SELECT product_cd FROM product WHERE name = 'savings account'),
 (SELECT cust_id FROM customer WHERE fed_id = '555-55-5555'),
  '2008-09-25', '2008-09-25', 'ACTIVE',
 (SELECT branch_id FROM branch WHERE name = 'Quincy Branch'),
 (SELECT emp_id FROM employee WHERE lname = 'Portman' AND fname = 'Frank'),
  0, 0);
```

读者使用单一 SQL 语句可以在 account 表里创建一行，同时查询 4 个外键值。不过，这种方法有一个缺点，就是当插入的列允许 null 值时，即使子查询不能返回值，insert 语句也会成功。例如，如果第四个子查询的 Frank Portman 的名字拼写错误，account 表中仍然会创建一个新行，但此时 open_emp_id 列的值被置为了 null。

9.6 子查询总结

本章涉及了非常多的内容，所以非常有必要再回顾一下。本章范例大致如下阐述了子查询：

- 它返回的结果可以是单行/单列，单列/多行以及多列/多行；
- 它可以独立于包含语句（非关联子查询）；
- 它可以引用包含语句中一行或多行（关联子查询）；
- 它可以用于条件中，这些条件使用比较运算符以及其他特殊目的的运算符（in、not in、 exists 和 not exists）；
- 它可以出现于 select、 update、 delete 和 insert 语句；
- 它产生的结果集可以与其他表或者子查询连接；
- 它可以生成值来填充表或者查询结果集中的一些列；
- 它可以用于查询中的 select、from、 where、 having 和 order by 子句。

显然，子查询是用途非常广泛的工具，所以在第一次读完本章后还没有了解所有概念时，也不必灰心。如果坚持不断地尝试使用子查询，那么读者很快就会发现每次写一个非平凡的 SQL 语句时都会考虑如何使用子查询来实现。

9.7 小测验

下面的习题测试读者对子查询的理解。完成后，可以参照附录 C 检查结果。

练习 9-1

对 account 表编写一个查询，过滤条件使用的非关联子查询实现对 product 表查找所有贷款账户（product.product_type_cd = 'LOAN'），结果包括账号 ID、产品代码、客户 ID 和可用余额。

练习 9-2

重做练习 9-1，对 product 表使用关联子查询获得同样的结果。

练习 9-3

将下面的查询与 employee 表连接，以展示每个雇员的经验。

```
SELECT 'trainee' name, '2004-01-01' start_dt, '2005-12-31' end_dt
UNION ALL
SELECT 'worker' name, '2002-01-01' start_dt, '2003-12-31' end_dt
UNION ALL
SELECT 'mentor' name, '2000-01-01' start_dt, '2001-12-31' end_dt
```

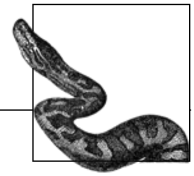
子查询别名定义为 levels，它包含雇员 ID、名字、姓氏以及经验等级（levels.name）。（提示：利用不等条件构建连接条件，确定 employee.start_date 列位于哪个等级。）

练习 9-4

对 employee 构建一个查询，检索雇员 ID、名字、姓氏及其所属部门和分行的名字。请不要连接任何表。

第 10 章

再谈连接



至此，在第 5 章中介绍的内连接这个概念，读者应该已经熟悉了。本章着重学习包括外连接和交叉连接在内的其他连接表的方法。

10.1 外连接

到目前为止，所列举的范例都包括多个表，此种情况下我没有考虑连接条件可能无法为表中的所有行匹配的问题。例如，当将 account 表与 customer 表连接时，我没有考虑可能出现 account 表中的 cust_id 列值无法匹配 customer 表中的 cust_id 列值。如果真是如此，则一个表或者其他表中的某些列就可能被遗漏在结果集之外。

要确定是否存在上述问题就必须检查表中的数据。下面查询 account 表中 account_id 列和 cust_id 列的值：

```
mysql> SELECT account_id, cust_id
    -> FROM account;
+------------+---------+
| account_id | cust_id |
+------------+---------+
|          1 |       1 |
|          2 |       1 |
|          3 |       1 |
|          4 |       2 |
|          5 |       2 |
|          7 |       3 |
|          8 |       3 |
|         10 |       4 |
|         11 |       4 |
|         12 |       4 |
|         13 |       5 |
|         14 |       6 |
```

```
|         15 |       6 |
|         17 |       7 |
|         18 |       8 |
|         19 |       8 |
|         21 |       9 |
|         22 |       9 |
|         23 |       9 |
|         24 |      10 |
|         25 |      10 |
|         27 |      11 |
|         28 |      12 |
|         29 |      13 |
+------------+---------+
24 rows in set (1.50 sec)
```

由查询结果可知，共 24 个账户，3 个不同客户，他们的 ID 为 1～13，每个客户至少有 1 个账户。下面是 customer 表中的客户 ID 集合：

```
mysql> SELECT cust_id
    -> FROM customer;
+---------+
| cust_id |
+---------+
|       1 |
|       2 |
|       3 |
|       4 |
|       5 |
|       6 |
|       7 |
|       8 |
|       9 |
|      10 |
|      11 |
|      12 |
|      13 |
+---------+
13 rows in set (0.02 sec)
```

由于 customer 表中的 ID 共有 13 行，分布在 1～13，因此 account 表中包含每个客户 ID 至少一次。所以当两个表在 cust_id 列连接时，期望的结果集应该是 24 行（不再添加其他过滤条件）。

```
mysql> SELECT a.acco unt_id, c.cust_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id;
+------------+---------+
| account_id | cust_id |
+------------+---------+
|          1 |       1 |
```

```
|          2 |       1 |
|          3 |       1 |
|          4 |       2 |
|          5 |       2 |
|          7 |       3 |
|          8 |       3 |
|         10 |       4 |
|         11 |       4 |
|         12 |       4 |
|         13 |       5 |
|         14 |       6 |
|         15 |       6 |
|         17 |       7 |
|         18 |       8 |
|         19 |       8 |
|         21 |       9 |
|         22 |       9 |
|         23 |       9 |
|         24 |      10 |
|         25 |      10 |
|         27 |      11 |
|         28 |      12 |
|         29 |      13 |
+------------+---------+
24 rows in set (0.06 sec)
```

正如所期望的那样，所有 24 个账户都出现在结果集中，但是若将 account 表连接到某一类客户表，比如 business 表，结果又会如何呢？

```
mysql> SELECT a.account_id, b.cust_id, b.name
    -> FROM account a INNER JOIN business b
    ->   ON a.cust_id = b.cust_id;
+------------+---------+------------------------+
| account_id | cust_id | name                   |
+------------+---------+------------------------+
|         24 |      10 | Chilton Engineering    |
|         25 |      10 | Chilton Engineering    |
|         27 |      11 | Northeast Cooling Inc  |
|         28 |      12 | Superior Auto Body     |
|         29 |      13 | AAA Insurance Inc.     |
+------------+---------+------------------------+
5 rows in set (0.10 sec)
```

现在的结果不是 24 行，而是仅有 5 行。让我们检查一下 business 表，看看为什么会这样：

```
mysql> SELECT cust_id, name
    -> FROM business;
+---------+------------------------+
| cust_id | name                   |
```

```
+---------+-----------------------+
|      10 | Chilton Engineering    |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body     |
|      13 | AAA Insurance Inc.     |
+---------+-----------------------+
4 rows in set (0.01 sec)
```

customer 表的 13 行中只有 4 个是商业客户，又由于其中一个商业客户有两个账户，所以 account 表中总共有 5 行连接到商业客户。

如果要查询返回所有账户，但只返回拥有这些账户的商业客户的名字，那该怎么办呢？下面的例子在 account 表和 business 表之间建立了外连接：

```
mysql> SELECT a.account_id, a.cust_id, b.name
    -> FROM account a LEFT OUTER JOIN business b
    ->   ON a.cust_id = b.cust_id;
+------------+---------+------------------------+
| account_id | cust_id | name                   |
+------------+---------+------------------------+
|          1 |       1 | NULL                   |
|          2 |       1 | NULL                   |
|          3 |       1 | NULL                   |
|          4 |       2 | NULL                   |
|          5 |       2 | NULL                   |
|          7 |       3 | NULL                   |
|          8 |       3 | NULL                   |
|         10 |       4 | NULL                   |
|         11 |       4 | NULL                   |
|         12 |       4 | NULL                   |
|         13 |       5 | NULL                   |
|         14 |       6 | NULL                   |
|         15 |       6 | NULL                   |
|         17 |       7 | NULL                   |
|         18 |       8 | NULL                   |
|         19 |       8 | NULL                   |
|         21 |       9 | NULL                   |
|         22 |       9 | NULL                   |
|         23 |       9 | NULL                   |
|         24 |      10 | Chilton Engineering    |
|         25 |      10 | Chilton Engineering    |
|         27 |      11 | Northeast Cooling Inc. |
|         28 |      12 |  Superior Auto Body    |
|         29 |      13 | AAA Insurance Inc.     |
+------------+---------+------------------------+
24 rows in set (0.04 sec)
```

外连接包括第一个表的所有行，但仅仅包含第二个表中那些匹配行的数据。在这种情况下，account 表的所有行都被包括了。由于指定了 left outer join ，因此 account 表就应

该在连接定义的左边。除了 4 个商业客户（cust_ids 为 10、11、12 和 13 的客户），所有行的 name 列值都是 null 。

```
mysql> SELECT a.account_id, a.cust_id, i.fname, i.lname
    -> FROM account a LEFT OUTER JOIN individual i
    ->   ON a.cust_id = i.cust_id;
+------------+---------+----------+---------+
| account_id | cust_id | fname    | lname   |
+------------+---------+----------+---------+
|          1 |       1 | James    | Hadley  |
|          2 |       1 | James    | Hadley  |
|          3 |       1 | James    | Hadley  |
|          4 |       2 | Susan    | Tingley |
|          5 |       2 | Susan    | Tingley |
|          7 |       3 | Frank    | Tucker  |
|          8 |       3 | Frank    | Tucker  |
|         10 |       4 | John     | Hayward |
|         11 |       4 | John     | Hayward |
|         12 |       4 | John     | Hayward |
|         13 |       5 | Charles  | Frasier |
|         14 |       6 | John     | Spencer |
|         15 |       6 | John     | Spencer |
|         17 |       7 | Margaret | Young   |
|         18 |       8 | George   | Blake   |
|         19 |       8 | George   | Blake   |
|         21 |       9 | Richard  | Farley  |
|         22 |       9 | Richard  | Farley  |
|         23 |       9 | Richard  | Farley  |
|         24 |      10 | NULL     | NULL    |
|         25 |      10 | NULL     | NULL    |
|         27 |      11 | NULL     | NULL    |
|         28 |      12 | NULL     | NULL    |
|         29 |      13 | NULL     | NULL    |
+------------+---------+----------+---------+
24 rows in set (0.09 sec)
```

这个查询内容实际上与前一个刚好相反：个人客户获得了名字和姓氏的信息，然而商业客户在这两列却是 null 。

10.1.1 左外连接与右外连接

在上面的所有外连接例子中，我使用了 left outer join。关键词 left 指出连接左边的表决定结果集的行数，而右边的只负责提供与之匹配的列值。不妨考虑如下查询：

```
mysql> SELECT c.cust_id, b.name
   -> FROM customer c LEFT OUTER JOIN business b
   ->   ON c.cust_id = b.cust_id;
+---------+------------------------+
| cust_id | name                   |
```

```
+---------+----------------------+
|       1 | NULL                 |
|       2 | NULL                 |
|       3 | NULL                 |
|       4 | NULL                 |
|       5 | NULL                 |
|       6 | NULL                 |
|       7 | NULL                 |
|       8 | NULL                 |
|       9 | NULL                 |
|      10 | Chilton Engineering  |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body   |
|      13 | AAA Insurance Inc.   |
+---------+----------------------+
13 rows in set (0.00 sec)
```

from 子句指定了一个左外连接，因此 customer 表的 13 行都包括在结果集中，而 business 表将 4 个商业客户的名字填入了结果集的第二列。如果指定的是 right outer join，那么结果如下：

```
mysql> SELECT c.cust_id, b.name
    -> FROM customer c RIGHT OUTER JOIN business b
    ->   ON c.cust_id = b.cust_id;
+---------+----------------------+
| cust_id | name                 |
+---------+----------------------+
|      10 | Chilton Engineering  |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body   |
|      13 | AAA Insurance Inc.   |
+---------+----------------------+
4 rows in set (0.00 sec)
```

现在结果集的行数由 business 表的行数决定，因此结果集只有 4 行。读者需要记住：这两个查询都是执行外连接，而关键字 left 和 right 只是通知服务器哪个表的数据可以不足。因此，若读者想要通过表 A 和 B 外连接得到结果为 A 中的所有行和 B 中匹配列的额外数据，则可以指定 A left outer join B 或者 B right outer join A。

10.1.2 三路外连接

有些情况下，读者可能想要将一个表与其他两个表进行外连接。比如，读者想要得到所有账户列表，其中包含个人客户的姓名以及商业客户的企业名称：

```
mysql> SELECT a.account_id, a.product_cd,
    ->   CONCAT(i.fname, ' ', i.lname) person_name,
    ->   b.name business_name
    -> FROM account a LEFT OUTER JOIN individual i
    ->   ON a.cust_id = i.cust_id
```

```
    -> LEFT OUTER JOIN business b
    -> ON a.cust_id = b.cust_id;
+------------+------------+-----------------+----------------------+
| account_id | product_cd | person_name     | business_name        |
+------------+------------+-----------------+----------------------+
|          1 | CHK        | James Hadley    | NULL                 |
|          2 | SAV        | James Hadley    | NULL                 |
|          3 | CD         | James Hadley    | NULL                 |
|          4 | CHK        | Susan Tingley   | NULL                 |
|          5 | SAV        | Susan Tingley   | NULL                 |
|          7 | CHK        | Frank Tucker    | NULL                 |
|          8 | MM         | Frank Tucker    | NULL                 |
|         10 | CHK        | John Hayward    | NULL                 |
|         11 | SAV        | John Hayward    | NULL                 |
|         12 | MM         | John Hayward    | NULL                 |
|         13 | CHK        | Charles Frasier | NULL                 |
|         14 | CHK        | John Spencer    | NULL                 |
|         15 | CD         | John Spencer    | NULL                 |
|         17 | CD         | Margaret Young  | NULL                 |
|         18 | CHK        | George Blake    | NULL                 |
|         19 | SAV        | George Blake    | NULL                 |
|         21 | CHK        | Richard Farley  | NULL                 |
|         22 | MM         | Richard Farley  | NULL                 |
|         23 | CD         | Richard Farley  | NULL                 |
|         24 | CHK        | NULL            | Chilton Engineering  |
|         25 | BUS        | NULL            | Chilton Engineering  |
|         27 | BUS        | NULL            | Northeast Cooling Inc. |
|         28 | CHK        | NULL            | Superior Auto Body   |
|         29 | SBL        | NULL            | AAA Insurance Inc.   |
+------------+------------+-----------------+----------------------+
24 rows in set (0.08 sec)
```

除了来自其他两个外连接表的个人姓名或企业名字，结果集包含了 account 表的所有 24 行。

我并不知道 MySQL 对于外连接到同一个表的其他表的数目是否有限制，不过我们总是可以利用子查询限制查询中连接的数目。例如，下面重写前一个例子：

```
mysql> SELECT account_ind.account_id, account_ind.product_cd,
    -> account_ind.person_name,
    -> b.name business_name
    -> FROM
    -> (SELECT a.account_id, a.product_cd, a.cust_id,
    ->   CONCAT(i.fname, ' ', i.lname) person_name
    -> FROM account a LEFT OUTER JOIN individual i
    ->   ON a.cust_id = i.cust_id) account_ind
    -> LEFT OUTER JOIN business b
    -> ON account_ind.cust_id = b.cust_id;
+------------+------------+-----------------+------------------------+
| account_id | product_cd | person_name     | business_name          |
```

```
+-----------+-----------+----------------+-----------------------+
|         1 | CHK       | James Hadley   | NULL                  |
|         2 | SAV       | James Hadley   | NULL                  |
|         3 | CD        | James Hadley   | NULL                  |
|         4 | CHK       | Susan Tingley  | NULL                  |
|         5 | SAV       | Susan Tingley  | NULL                  |
|         7 | CHK       | Frank Tucker   | NULL                  |
|         8 | MM        | Frank Tucker   | NULL                  |
|        10 | CHK       | John Hayward   | NULL                  |
|        11 | SAV       | John Hayward   | NULL                  |
|        12 | MM        | John Hayward   | NULL                  |
|        13 | CHK       | Charles Frasier | NULL                 |
|        14 | CHK       | John Spencer   | NULL                  |
|        15 | CD        | John Spencer   | NULL                  |
|        17 | CD        | Margaret Young | NULL                  |
|        18 | CHK       | George Blake   | NULL                  |
|        19 | SAV       | George Blake   | NULL                  |
|        21 | CHK       | Richard Farley | NULL                  |
|        22 | MM        | Richard Farley | NULL                  |
|        23 | CD        | Richard Farley | NULL                  |
|        24 | CHK       | NULL           | Chilton Engineering   |
|        25 | BUS       | NULL           | Chilton Engineering   |
|        27 | BUS       | NULL           | Northeast Cooling Inc. |
|        28 | CHK       | NULL           | Superior Auto Body    |
|        29 | SBL       | NULL           | AAA Insurance Inc.    |
+-----------+-----------+----------------+-----------------------+
24 rows in set (0.08 sec)
```

在这个版本的查询中，子查询 account_ind 将 individual 表外连接到 account 表，而其结果又外连接到 business 表。由此可见，每个查询（子查询和包含查询）只使用了单一外连接。读者使用的数据库如果不是 MySQL，那么可能需要采取这种策略来实现多表的外连接。

10.1.3 自外连接

在第 5 章中，我介绍了自连接的概念，即一个表连接自己。下面是第 5 章里的一个自连接范例，它将 employee 表连接到自己而生成雇员和他们主管的列表：

```
mysql> SELECT e.fname, e.lname,
    ->   e_mgr.fname mgr_fname, e_mgr.lname mgr_lname
    -> FROM employee e INNER JOIN employee e_mgr
    ->   ON e.superior_emp_id = e_mgr.emp_id;
+----------+-----------+-----------+-----------+
| fname    | lname     | mgr_fname | mgr_lname |
+----------+-----------+-----------+-----------+
| Susan    | Barker    | Michael   | Smith     |
| Robert   | Tyler     | Michael   | Smith     |
| Susan    | Hawthorne | Robert    | Tyler     |
| John     | Gooding   | Susan     | Hawthorne |
```

```
| Helen    | Fleming   | Susan     | Hawthorne |
| Chris    | Tucker    | Helen     | Fleming   |
| Sarah    | Parker    | Helen     | Fleming   |
| Jane     | Grossman  | Helen     | Fleming   |
| Paula    | Roberts   | Susan     | Hawthorne |
| Thomas   | Ziegler   | Paula     | Roberts   |
| Samantha | Jameson   | Paula     | Roberts   |
| John     | Blake     | Susan     | Hawthorne |
| Cindy    | Mason     | John      | Blake     |
| Frank    | Portman   | John      | Blake     |
| Theresa  | Markham   | Susan     | Hawthorne |
| Beth     | Fowler    | Theresa   | Markham   |
| Rick     | Tulman    | Theresa   | Markham   |
+----------+-----------+-----------+-----------+
17 rows in set (0.02 sec)
```

这个查询运行的很好，但是存在一个小问题：没有主管的雇员将被遗漏到结果集之外。如果将内连接变为外连接，那么结果集就包括所有雇员，当然没有主管的雇员也被包括在内：

```
mysql> SELECT e.fname, e.lname,
    ->   e_mgr.fname mgr_fname, e_mgr.lname mgr_lname
    -> FROM employee e LEFT OUTER JOIN employee e_mgr
    ->   ON e.superior_emp_id = e_mgr.emp_id;
+----------+-----------+-----------+-----------+
| fname    | lname     | mgr_fname | mgr_lname |
+----------+-----------+-----------+-----------+
| Michael  | Smith     | NULL      | NULL      |
| Susan    | Barker    | Michael   | Smith     |
| Robert   | Tyler     | Michael   | Smith     |
| Susan    | Hawthorne | Robert    | Tyler     |
| John     | Gooding   | Susan     | Hawthorne |
| Helen    | Fleming   | Susan     | Hawthorne |
| Chris    | Tucker    | Helen     | Fleming   |
| Sarah    | Parker    | Helen     | Fleming   |
| Jane     | Grossman  | Helen     | Fleming   |
| Paula    | Roberts   | Susan     | Hawthorne |
| Thomas   | Ziegler   | Paula     | Roberts   |
| Samantha | Jameson   | Paula     | Roberts   |
| John     | Blake     | Susan     | Hawthorne |
| Cindy    | Mason     | John      | Blake     |
| Frank    | Portman   | John      | Blake     |
| Theresa  | Markham   | Susan     | Hawthorne |
| Beth     | Fowler    | Theresa   | Markham   |
| Rick     | Tulman    | Theresa   | Markham   |
+----------+-----------+-----------+-----------+
18 rows in set (0.00 sec)
```

现在结果集中包括银行总裁 Michael Smith，而他没有主管。查询使用左外连接生成所

有雇员以及可能的主管。如果将这个连接改为右外连接，那么情况如下：

```
mysql> SELECT e.fname, e.lname,
    ->   e_mgr.fname mgr_fname, e_mgr.lname mgr_lname
    -> FROM employee e RIGHT OUTER JOIN employee e_mgr
    ->  ON e.superior_emp_id = e_mgr.emp_id;
+----------+----------+-----------+-----------+
| fname    | lname    | mgr_fname | mgr_lname |
+----------+----------+-----------+-----------+
| Susan    | Barker   | Michael   | Smith     |
| Robert   | Tyler    | Michael   | Smith     |
| NULL     | NULL     | Susan     | Barker    |
| Susan    | Hawthorne | Robert   | Tyler     |
| John     | Gooding  | Susan     | Hawthorne |
| Helen    | Fleming  | Susan     | Hawthorne |
| Paula    | Roberts  | Susan     | Hawthorne |
| John     | Blake    | Susan     | Hawthorne |
| Theresa  | Markham  | Susan     | Hawthorne |
| NULL     | NULL     | John      | Gooding   |
| Chris    | Tucker   | Helen     | Fleming   |
| Sarah    | Parker   | Helen     | Fleming   |
| Jane     | Grossman | Helen     | Fleming   |
| NULL     | NULL     | Chris     | Tucker    |
| NULL     | NULL     | Sarah     | Parker    |
| NULL     | NULL     | Jane      | Grossman  |
| Thomas   | Ziegler  | Paula     | Roberts   |
| Samantha | Jameson  | Paula     | Roberts   |
| NULL     | NULL     | Thomas    | Ziegler   |
| NULL     | NULL     | Samantha  | Jameson   |
| Cindy    | Mason    | John      | Blake     |
| Frank    | Portman  | John      | Blake     |
| NULL     | NULL     | Cindy     | Mason     |
| NULL     | NULL     | Frank     | Portman   |
| Beth     | Fowler   | Theresa   | Markham   |
| Rick     | Tulman   | Theresa   | Markham   |
| NULL     | NULL     | Beth      | Fowler    |
| NULL     | NULL     | Rick      | Tulman    |
+----------+----------+-----------+-----------+
28 rows in set (0.00 sec)
```

这个查询显示的内容除了每个主管（仍然是第三行和第四行）外，还有其管理的雇员集合。因此，Michael Smith 作为 Susan Barker 和 Robert Tyler 的主管出现了两次，而 Susan Barker 显示了一次，但她不是任何人的主管（因为该行的第一列和第二列值为 null）。所有 18 个雇员在第三列和第四列出现至少一次，其中管理雇员超过一人的会出现不止一次，因此结果集共有 28 行。这与上一个查询明显不同，而这仅仅是由改变一个关键字（从 left 到 right）引起的。所以，读者在使用外连接时要确定到底使用左外连接还是右外连接。

10.2 交叉连接

早在第 5 章中就介绍了笛卡儿积的概念，本质上它就是在不指定任何连接条件的情况下多表连接的结果。笛卡儿积在发生意外时使用相当频繁（比如，忘记在 from 子句中添加连接条件），否则并不会被经常使用。如果读者确实打算生成两个表的笛卡儿积，那么需要指定交叉连接，例如：

```
mysql> SELECT pt.name, p.product_cd, p.name
    -> FROM product p CROSS JOIN product_type pt;
+-------------------------------+------------+-------------------------+
| name                          | product_cd | name                    |
+-------------------------------+------------+-------------------------+
| Customer Accounts             | AUT        | auto loan               |
| Customer Accounts             | BUS        | business line of credit |
| Customer Accounts             | CD         | certificate of deposit  |
| Customer Accounts             | CHK        | checking account        |
| Customer Accounts             | MM         | money market account    |
| Customer Accounts             | MRT        | home mortgage           |
| Customer Accounts             | SAV        | savings account         |
| Customer Accounts             | SBL        | small business loan     |
| Insurance Offerings           | AUT        | auto loan               |
| Insurance Offerings           | BUS        | business line of credit |
| Insurance Offerings           | CD         | certificate of deposit  |
| Insurance Offerings           | CHK        | checking account        |
| Insurance Offerings           | MM         | money market account    |
| Insurance Offerings           | MRT        | home mortgage           |
| Insurance Offerings           | SAV        | savings account         |
| Insurance Offerings           | SBL        | small business loan     |
| Individual and Business Loans | AUT        | auto loan               |
| Individual and Business Loans | BUS        | business line of credit |
| Individual and Business Loans | CD         | certificate of deposit  |
| Individual and Business Loans | CHK        | checking account        |
| Individual and Business Loans | MM         | money market account    |
| Individual and Business Loans | MRT        | home mortgage           |
| Individual and Business Loans | SAV        | savings account         |
| Individual and Business Loans | SBL        | small business loan     |
+-------------------------------+------------+-------------------------+
24 rows in set (0.00 sec)
```

这个查询生成了 product 表和 product_type 表的笛卡儿积，结果集有 24 行（product 表有 8 行， product_type 表有 3 行）。现在读者知道交叉连接是什么、如何指定它、为什么要使用它了吗？大多数 SQL 书籍都会先描述什么是交叉连接，然后告诉读者它很少被使用。不过，现在我想分享非常适合使用交叉连接的情况。

在第 9 章中，我讨论了如何使用子查询创建虚拟表。我使用的例子展示了如何构建可以与其他表连接的 3 行表。下面是例子的虚拟表：

```
mysql> SELECT 'Small Fry' name, 0 low_limit, 4999.99 high_limit
    -> UNION ALL
    -> SELECT 'Average Joes' name, 5000 low_limit, 9999.99 high_limit
    -> UNION ALL
    -> SELECT 'Heavy Hitters' name, 10000 low_limit, 9999999.99 high_limit;
+---------------+-----------+------------+
| name          | low_limit | high_limit |
+---------------+-----------+------------+
| Small Fry     |         0 |    4999.99 |
| Average Joes  |      5000 |    9999.99 |
| Heavy Hitters |     10000 | 9999999.99 |
+---------------+-----------+------------+
3 rows in set (0.00 sec)
```

虽然这个表正是将客户依据其总账余额分为 3 组所需要的，但是这种使用集合运算符 union all 合并单行表的策略在虚拟构建一个大型表时就不能很好地起作用了。

例如，读者打算创建一个查询，为 2008 年每天生成一行，但是数据库中没有包含每天一行的表。读者如果使用第 9 章中的策略，就可能像下面这样做：

```
SELECT '2008-01-01' dt
UNION ALL
SELECT '2008-01-02' dt
UNION ALL
SELECT '2008-01-03' dt
UNION ALL
...
...
...
SELECT '2008-12-29' dt
UNION ALL
SELECT '2008-12-30' dt
UNION ALL
SELECT '2008-12-31' dt
```

构建一个查询来将 366 个查询的结果合并到一起确实有点单调乏味，因此可能需要一个不同的策略。或许可以这样：首先通过单列生成 366 行的表，此单列包含 0～366 中的某个数字（2008 年是闰年），然后将这个天数加上 2008 年 1 月 1 日，这种思路如何呢？下面是一种可能生成这样一个表的方法：

```
mysql> SELECT ones.num + tens.num + hundreds.num
    -> FROM
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 1 num UNION ALL
    ->   SELECT 2 num UNION ALL
    ->   SELECT 3 num UNION ALL
    ->   SELECT 4 num UNION ALL
    ->   SELECT 5 num UNION ALL
    ->   SELECT 6 num UNION ALL
```

```
    ->   SELECT 7 num UNION ALL
    ->   SELECT 8 num UNION ALL
    ->   SELECT 9 num) ones
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 10 num UNION ALL
    ->   SELECT 20 num UNION ALL
    ->   SELECT 30 num UNION ALL
    ->   SELECT 40 num UNION ALL
    ->   SELECT 50 num UNION ALL
    ->   SELECT 60 num UNION ALL
    ->   SELECT 70 num UNION ALL
    ->   SELECT 80 num UNION ALL
    ->   SELECT 90 num) tens
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 100 num UNION ALL
    ->   SELECT 200 num UNION ALL
    ->   SELECT 300 num) hundreds;
+-------------------------------------+
| ones.num + tens.num + hundreds.num |
+-------------------------------------+
|                                   0 |
|                                   1 |
|                                   2 |
|                                   3 |
      |                             4 |
|                                   5 |
|                                   6 |
|                                   7 |
|                                   8 |
|                                   9 |
|                                  10 |
|                                  11 |
|                                  12 |
...
...
...
|                                 391 |
|                                 392 |
|                                 393 |
|                                 394 |
|                                 395 |
|                                 396 |
|                                 397 |
|                                 398 |
|                                 399 |
+-------------------------------------+
400 rows in set (0.00 sec)
```

如果生成{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}、{0, 10, 20, 30, 40, 50, 60, 70, 80, 90}和 {0, 100, 200, 300}这 3 个集合的笛卡儿积，并将这 3 列相加，那么就能得到包含 0～399 所有数字的 400 行结果集。可是这超过了生成 2008 年日期集所需的 366 行，没有关系，消除这些额外的行很容易，我随后将会介绍如何去做。

下一步是将数字集转换为日期集。为此，我需要首先使用 date_add()函数将结果集中的数字加上 2008 年 1 月 1 日，然后添加一个过滤条件来排除超过 2009 年的所有日期。

```
mysql> SELECT DATE_ADD('2008-01-01',
    ->   INTERVAL (ones.num + tens.num + hundreds.num) DAY) dt
    -> FROM
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 1 num UNION ALL
    ->   SELECT 2 num UNION ALL
    ->   SELECT 3 num UNION ALL
    ->   SELECT 4 num UNION ALL
    ->   SELECT 5 num UNION ALL
    ->   SELECT 6 num UNION ALL
    ->   SELECT 7 num UNION ALL
    ->   SELECT 8 num UNION ALL
    ->   SELECT 9 num) ones
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 10 num UNION ALL
    ->   SELECT 20 num UNION ALL
    ->   SELECT 30 num UNION ALL
    ->   SELECT 40 num UNION ALL
    ->   SELECT 50 num UNION ALL
    ->   SELECT 60 num UNION ALL
    ->   SELECT 70 num UNION ALL
    ->   SELECT 80 num UNION ALL
    ->   SELECT 90 num) tens
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 100 num UNION ALL
    ->   SELECT 200 num UNION ALL
    ->   SELECT 300 num) hundreds
    -> WHERE DATE_ADD('2008-01-01',
    ->   INTERVAL (ones.num + tens.num + hundreds.num) DAY) < '2009-01-01'
    -> ORDER BY 1;
+------------+
| dt         |
+------------+
| 2008-01-01 |
| 2008-01-02 |
| 2008-01-03 |
| 2008-01-04 |
| 2008-01-05 |
```

```
| 2008-01-06 |
| 2008-01-07 |
| 2008-01-08 |
| 2008-01-09 |
| 2008-01-10 |
...
...
...
| 2008-02-20 |
| 2008-02-21 |
| 2008-02-22 |
| 2008-02-23 |
| 2008-02-24 |
| 2008-02-25 |
| 2008-02-26 |
| 2008-02-27 |
| 2008-02-28 |
| 2008-02-29 |
| 2008-03-01 |
...
...
...
| 2008-12-20 |
| 2008-12-21 |
| 2008-12-22 |
| 2008-12-23 |
| 2008-12-24 |
| 2008-12-25 |
| 2008-12-26 |
| 2008-12-27 |
| 2008-12-28 |
| 2008-12-29 |
| 2008-12-30 |
| 2008-12-31 |
+------------+
366 rows in set (0.01 sec)
```

这个方法的妙处在于无需读者介入，结果集会自动包含额外的闰日（2 月 29 日），当然这是由数据库服务器将 2008 年 1 月 1 日加上 59 日计算出来的。

现在有了构造 2008 年所有日子的方法，那么如何使用它呢？又如何生成一个查询来展示 2008 年每一日、当天的银行交易数量以及开户数量等数据呢？下面的例子解答了这个问题：

```
mysql> SELECT days.dt, COUNT(t.txn_id)
    -> FROM transaction t RIGHT OUTER JOIN
    ->  (SELECT DATE_ADD('2008-01-01',
    ->     INTERVAL (ones.num + tens.num + hundreds.num) DAY) dt
    ->   FROM
```

```
    ->   (SELECT 0 num UNION ALL
    ->    SELECT 1 num UNION ALL
    ->    SELECT 2 num UNION ALL
    ->    SELECT 3 num UNION ALL
    ->    SELECT 4 num UNION ALL
    ->    SELECT 5 num UNION ALL
    ->    SELECT 6 num UNION ALL
    ->    SELECT 7 num UNION ALL
    ->    SELECT 8 num UNION ALL
    ->    SELECT 9 num) ones
    ->    CROSS JOIN
    ->   (SELECT 0 num UNION ALL
    ->    SELECT 10 num UNION ALL
    ->    SELECT 20 num UNION ALL
    ->    SELECT 30 num UNION ALL
    ->    SELECT 40 num UNION ALL
    ->    SELECT 50 num UNION ALL
    ->    SELECT 60 num UNION ALL
    ->    SELECT 70 num UNION ALL
    ->    SELECT 80 num UNION ALL
    ->    SELECT 90 num) tens
    ->    CROSS JOIN
    ->   (SELECT 0 num UNION ALL
    ->    SELECT 100 num UNION ALL
    ->    SELECT 200 num UNION ALL
    ->    SELECT 300 num) hundreds
    ->  WHERE DATE_ADD('2008-01-01',
    ->    INTERVAL (ones.num + tens.num + hundreds.num) DAY) <
    ->      '2009-01-01') days
    ->  ON days.dt = t.txn_date
    -> GROUP BY days.dt
    -> ORDER BY 1;
+------------+-----------------+
| dt         | COUNT(t.txn_id) |
+------------+-----------------+
| 2008-01-01 |               0 |
| 2008-01-02 |               0 |
| 2008-01-03 |               0 |
| 2008-01-04 |               0 |
| 2008-01-05 |              21 |
| 2008-01-06 |               0 |
| 2008-01-07 |               0 |
| 2008-01-08 |               0 |
| 2008-01-09 |               0 |
| 2008-01-10 |               0 |
| 2008-01-11 |               0 |
| 2008-01-12 |               0 |
| 2008-01-13 |               0 |
| 2008-01-14 |               0 |
```

```
| 2008-01-15 |                0 |
...
| 2008-12-31 |                0 |
+------------+-----------------+
366 rows in set (0.03 sec)
```

这是至此本书中最有趣的查询之一，其中包括交叉连接、外连接、日期函数、分组、集合运算符（union all）以及聚合函数（count()）。当然，这并不是所给问题的最优雅的解决方法，但是它给了我们一些启迪：即使对交叉连接这样很少使用的功能，也可以通过一点创造力和对语言本身的牢牢把握使其成为读者 SQL 工具包中一个强有力的工具。

10.3 自然连接

读者如果想少动一点脑筋，就可以选择这样一个连接类型：自己指定相关的表，而让数据库服务器去决定需要什么样的连接条件。所谓自然连接，是指依赖多表交叉时的相同列名来推断正确的连接条件。例如，account 表包含了一个名为 cust_id 的列，它是来自 customer 表的外键，也是 customer 表的主键。读者可以使用自然连接编写一个查询来连接这两个表：

```
mysql> SELECT a.account_id, a.cust_id, c.cust_type_cd, c.fed_id
    -> FROM account a NATURAL JOIN customer c;
+------------+---------+--------------+-------------+
| account_id | cust_id | cust_type_cd | fed_id      |
+------------+---------+--------------+-------------+
|          1 |       1 | I            | 111-11-1111 |
|          2 |       1 | I            | 111-11-1111 |
|          3 |       1 | I            | 111-11-1111 |
|          4 |       2 | I            | 222-22-2222 |
|          5 |       2 | I            | 222-22-2222 |
|          6 |       3 | I            | 333-33-3333 |
|          7 |       3 | I            | 333-33-3333 |
|          8 |       4 | I            | 444-44-4444 |
|          9 |       4 | I            | 444-44-4444 |
|         10 |       4 | I            | 444-44-4444 |
|         11 |       5 | I            | 555-55-5555 |
|         12 |       6 | I            | 666-66-6666 |
|         13 |       6 | I            | 666-66-6666 |
|         14 |       7 | I            | 777-77-7777 |
|         15 |       8 | I            | 888-88-8888 |
|         16 |       8 | I            | 888-88-8888 |
|         17 |       9 | I            | 999-99-9999 |
|         18 |       9 | I            | 999-99-9999 |
|         19 |       9 | I            | 999-99-9999 |
|         20 |      10 | B            | 04-1111111  |
|         21 |      10 | B            | 04-1111111  |
```

```
|         22 |       11 | B            | 04-2222222  |
|         23 |       12 | B            | 04-3333333  |
|         24 |       13 | B            | 04-4444444  |
+------------+---------+--------------+-------------+
24 rows in set (0.02 sec)
```

由于指定了自然连接，因此服务器检查表的定义并给两个表的连接添加了连接条件 a.cust_id = c.cust_id。

这一切都很好，但是如果交叉表没有相同名称的列怎么办？例如，account 表有来自 branch 表的外键，但是在 account 表中命名为 open_branch_id 而不是 branch_id。下面看看在 account 表和 branch 表之间使用自然连接会如何：

```
mysql> SELECT a.account_id, a.cust_id, a.open_branch_id, b.name
    -> FROM account a NATURAL JOIN branch b;
+------------+---------+----------------+--------------+
| account_id | cust_id | open_branch_id | name         |
+------------+---------+----------------+--------------+
|          1 |       1 |              2 | Headquarters |
|          1 |       1 |              2 | Woburn Branch |
|          1 |       1 |              2 | Quincy Branch |
|          1 |       1 |              2 | So. NH Branch |
|          2 |       1 |              2 | Headquarters |
|          2 |       1 |              2 | Woburn Branch |
|          2 |       1 |              2 | Quincy Branch |
|          2 |       1 |              2 | So. NH Branch |
|          3 |       1 |              2 | Headquarters |
|          3 |       1 |              2 | Woburn Branch |
|          3 |       1 |              2 | Quincy Branch |
|          3 |       1 |              2 | So. NH Branch |
|          4 |       2 |              2 | Headquarters |
|          4 |       2 |              2 | Woburn Branch |
|          4 |       2 |              2 | Quincy Branch |
|          4 |       2 |              2 | So. NH Branch |
|          5 |       2 |              2 | Headquarters |
|          5 |       2 |              2 | Woburn Branch |
|          5 |       2 |              2 | Quincy Branch |
|          5 |       2 |              2 | So. NH Branch |
|          7 |       3 |              3 | Headquarters |
|          7 |       3 |              3 | Woburn Branch |
|          7 |       3 |              3 | Quincy Branch |
|          7 |       3 |              3 | So. NH Branch |
|          8 |       3 |              3 | Headquarters |
|          8 |       3 |              3 | Woburn Branch |
|          8 |       3 |              3 | Quincy Branch |
|          8 |       3 |              3 | So. NH Branch |
|         10 |       4 |              1 | Headquarters |
|         10 |       4 |              1 | Woburn Branch |
|         10 |       4 |              1 | Quincy Branch |
```

```
|         10 |       4 |                1 | So. NH Branch |
...
...
...
|         24 |      10 |                4 | Headquarters  |
|         24 |      10 |                4 | Woburn Branch |
|         24 |      10 |                4 | Quincy Branch |
|         24 |      10 |                4 | So. NH Branch |
|         25 |      10 |                4 | Headquarters  |
|         25 |      10 |                4 | Woburn Branch |
|         25 |      10 |                4 | Quincy Branch |
|         25 |      10 |                4 | So. NH Branch |
|         27 |      11 |                2 | Headquarters  |
|         27 |      11 |                2 | Woburn Branch |
|         27 |      11 |                2 | Quincy Branch |
|         27 |      11 |                2 | So. NH Branch |
|         28 |      12 |                4 | Headquarters  |
|         28 |      12 |                4 | Woburn Branch |
|         28 |      12 |                4 | Quincy Branch |
|         28 |      12 |                4 | So. NH Branch |
|         29 |      13 |                3 | Headquarters  |
|         29 |      13 |                3 | Woburn Branch |
|         29 |      13 |                3 | Quincy Branch |
|         29 |      13 |                3 | So. NH Branch |
+------------+---------+-----------------+---------------+
96 rows in set (0.07 sec)
```

看上去似乎出了一点问题，这个查询不应该返回超过 24 行的结果，因为 account 表只有 24 行。究竟发生了什么问题？原来是服务器由于找不到两个相同名字的列而不能生成连接条件，只好交叉连接两表，所以最终结果为 96 行（24 个账户，4 个分行）。

仅仅为不必输入连接条件而省点事却产生这种麻烦值不值得？绝对不值得！读者应该避免这种连接类型，而使用有明确连接条件的内连接。

10.4 小测验

下面的习题测试读者对于外连接和交叉连接的理解。完成习题后，请参考附录 C 检查答案。

练习 10-1

编写一个查询，它返回所有产品名称及基于该产品的账号（用 account 表里的 product_cd 列连接 product 表）。查询结果需要包括所有产品，即使这个产品并没有客户开户。

练习 10-2

利用其他外连接类型重写练习 10-1 的查询（比如，若在练习 10-1 中使用了左外连接，

这次就使用右外连接），要求查询结果相同。

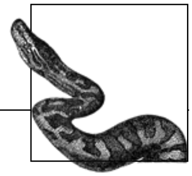
练习 10-3

编写一个查询，将 account 表与 individual 和 business 两个表外连接（通过 account.cust_id 列）。要求结果集中每个账户一行，查询的列有 account.account_id、account.product_cd、individual.fname、individual.lname 和 business.name。

练习 10-4（附加题）

设计一个查询，生成集合{1, 2, 3…, 99, 100}。（提示：应用交叉连接，至少有两个 from 子句的子查询。）

第 11 章



条件逻辑

在某些情况下，读者需要的 SQL 逻辑分支在这个方向还是另一个方向取决于特定列或者表达式的值。本章着重讨论如何写一个语句使之随着遇到的数据的不同而采取不同的执行方式。

11.1　什么是条件逻辑

简单地说，条件逻辑是程序执行时从多个路径中选取其一的能力。例如，查询客户信息时，读者可能希望依据客户类型决定从 individual 表中检索 fname/lname 列或者从 business 表中检索 name 列。使用外连接，可以返回两个字符，然后再让读者找出该使用哪个，具体如下：

```
mysql> SELECT c.cust_id, c.fed_id, c.cust_type_cd,
    ->   CONCAT(i.fname, ' ', i.lname) indiv_name,
    ->   b.name business_name
    -> FROM customer c LEFT OUTER JOIN individual i
    ->   ON c.cust_id = i.cust_id
    ->   LEFT OUTER JOIN business b
    ->   ON c.cust_id = b.cust_id;
+---------+------------+------------+---------------+-----------+
| cust_id | fed_id     | cust_type_cd | indiv_name      |business_name |
+---------+------------+------------+---------------+-----------+
|      1  |111-11-1111  | I            | James Hadley    | NULL         |
|      2  |222-22-2222  | I            | Susan Tingley   | NULL         |
|      3  |333-33-3333  | I            | Frank Tucker    | NULL         |
|      4  |444-44-4444  | I            | John Hayward    | NULL         |
|      5  |555-55-5555  | I            | Charles Frasier | NULL         |
|      6  |666-66-6666  | I            | John Spencer    | NULL         |
|      7  |777-77-7777  | I            | Margaret Young  | NULL         |
|      8  |888-88-8888  | I            | Louis Blake     | NULL         |
```

```
|      9    |999-99-9999    | I             | Richard Farley    | NULL          |
|     10    |04-1111111     | B             | NULL              | Chilton Engin eering |
|     11    | 04-2222222    | B             | NULL              |Northeast Cooling Inc |
|     12    | 04-3333333    | B             | NULL              | Superior Auto Body |
|     13    | 04-4444444    | B             | NULL              | AAA Insurance Inc. |
+---------+------------+------------+---------------+------------+
13 rows in set (0.13 sec)
```

读者可以先查看 cust_type_cd 列的值，然后决定是使用 indiv_name 还是 business_name 列的值。还有另一种方法，读者可以通过 case 表达式使用条件逻辑决定客户类型，进而返回恰当的字符串：

```
mysql> SELECT c.cust_id, c.fed_id,
    ->   CASE
    ->     WHEN c.cust_type_cd = 'I'
    ->       THEN CONCAT(i.fname, ' ', i.lname)
    ->     WHEN c.cust_type_cd = 'B'
    ->       THEN b.name
    ->     ELSE 'Unknown'
    ->   END name
    -> FROM customer c LEFT OUTER JOIN individual i
    ->   ON c.cust_id = i.cust_id
    ->   LEFT OUTER JOIN business b
    ->   ON c.cust_id = b.cust_id;
+---------+------------+------------------------+
| cust_id | fed_id     | name                   |
+---------+------------+------------------------+
|       1 | 111-11-1111 | James Hadley          |
|       2 | 222-22-2222 | Susan Tingley         |
|       3 | 333-33-3333 | Frank Tucker          |
|       4 | 444-44-4444 | John Hayward          |
|       5 | 555-55-5555 | Charles Frasier       |
|       6 | 666-66-6666 | John Spencer          |
|       7 | 777-77-7777 | Margaret Young        |
|       8 | 888-88-8888 | Louis Blake           |
|       9 | 999-99-9999 | Richard Farley        |
|      10 | 04-1111111  | Chilton Engineering   |
|      11 | 04-2222222  | Northeast Cooling Inc. |
|      12 | 04-3333333  | Superior Auto Body    |
|      13 | 04-4444444  | AAA Insurance Inc.    |
+---------+------------+------------------------+
13 rows in set (0.00 sec)
```

这个版本的查询只返回由 case 表达式生成的单个 name 列。这个从查询的第二行起的 case 表达式首先检查 cust_type_cd 列的值，然后依据该值决定返回个人姓名还是企业名称。

11.2 case 表达式

大部分编程语言中都存在 if-then-else 语句，而所有的主流服务器也都包含模拟此功能的内置函数（比如Oracle的decode()函数，MySQL的 if()函数以及SQL Server的 coalesce()函数）：

- case 表达式是 SQL 标准的一部分（SQL92 发布），并且已在 Oracle 数据库、SQL Server、MySQL、Sybase、PostgreSQL、IBM UDB 及其他数据库中实现；
- case 表达式已经内置于 SQL 语法，可以用于 select、insert、update 和 delete 语句。

下面两小节介绍两种不同类型的 case 表达式，随后我将展示一些实战中的 case 表达式范例。

11.2.1 查找型 case 表达式

本章前面展示的 case 表达式是一种查找型表达式的范例，其语法如下：

```
CASE
  WHEN C1 THEN E1
  WHEN C2 THEN E2
  ...
  WHEN CN THEN EN
  [ELSE ED]
END
```

在上面的定义中，符号 C1、 C2、...、CN 代表条件， E1 、E2、…、EN 代表 case 表达式返回的表达式。如果 when 子句中条件为真，那么 case 表达式返回相应的表达式。另外，符号 ED 代表默认表达式，也就是 case 表达式在条件 C1、 C2、…、CN 中没有一个为真时将返回的表达式（else 子句是可选的，这就是为什么要用方括号把它括起来）。各种各样的 when 子句返回的所有表达式的计算结果必须类型相同（如日期型、数字型、变长字符串型等）。

下面是一个查找型 case 表达式范例：

```
CASE
  WHEN employee.title = 'Head Teller'
    THEN 'Head Teller'
  WHEN employee.title = 'Teller'
    AND YEAR(employee.start_date) > 2007
    THEN 'Teller Trainee'
  WHEN employee.title = 'Teller'
    AND YEAR(employee.start_date) < 2006
    THEN 'Experienced Teller'
  WHEN employee.title = 'Teller'
    THEN 'Teller'
```

```
  ELSE 'Non-Teller'
END
```

case 表达式返回的字符串可以用来决定薪级、打印名片等。case 表达式开始执行时，when 子句会从上到下地执行，只要有一个 when 子句值为真，就会返回相应的表达式，同时忽略其他 when 子句。如果没有一个 when 子句条件的值为真，那么将会返回 else 子句里的表达式。

虽然上面的例子返回的是字符串表达式，但是事实上 case 表达式可以返回任意类型的表达式，甚至包括子查询。下面是本章前面所介绍的个人姓名/企业名字查询的另一种版本，这一次使用子查询代替外连接从 individual 和 business 表中检索数据：

```
mysql> SELECT c.cust_id, c.fed_id,
    ->   CASE
    ->     WHEN c.cust_type_cd = 'I' THEN
    ->      (SELECT CONCAT(i.fname, ' ', i.lname)
    ->       FROM individual i
    ->       WHERE i.cust_id = c.cust_id)
    ->     WHEN c.cust_type_cd = 'B' THEN
    ->      (SELECT b.name
    ->       FROM business b
    ->       WHERE b.cust_id = c.cust_id)
    ->     ELSE 'Unknown'
    ->   END name
    -> FROM customer c;
+---------+-------------+------------------------+
| cust_id | fed_id      | name                   |
+---------+-------------+------------------------+
|       1 | 111-11-1111 | James Hadley           |
|       2 | 222-22-2222 | Susan Tingley          |
|       3 | 333-33-3333 | Frank Tucker           |
|       4 | 444-44-4444 | John Hayward           |
|       5 | 555-55-5555 | Charles Frasier        |
|       6 | 666-66-6666 | John Spencer           |
|       7 | 777-77-7777 | Margaret Young         |
|       8 | 888-88-8888 | Louis Blake            |
|       9 | 999-99-9999 | Richard Farley         |
|      10 | 04-1111111  | Chilton Engineering    |
|      11 | 04-2222222  | Northeast Cooling Inc. |
|      12 | 04-3333333  | Superior Auto Body     |
|      13 | 04-4444444  | AAA Insurance Inc.     |
+---------+-------------+------------------------+
13 rows in set (0.01 sec)
```

这个版本的查询只包括 from 子句中的 customer 表，并且使用关联查询为每个客户检索

合适的姓名。相较于本章前面的外连接版本，我更喜欢这个版本，因为使用它服务器只在需要时才从 individual 和 business 表中读取数据，而不会总是连接所有的表。

11.2.2　简单 case 表达式

简单 case 表达式和查找型 case 表达式非常相似，但灵活性不够，其语法如下：

```
CASE V0
  WHEN V1 THEN E1
  WHEN V2 THEN E2
  ...
  WHEN VN THEN EN
  [ELSE ED]
END
```

在前面的定义中， V0 代表一个值，符号 V1、V2、…、VN 代表要与 V0 比较的值，符号 E1、E2、…、EN 代表 case 表达式要返回的表达式， ED 代表 V1、 V2、…、VN 没有一个值匹配 V0 时返回的默认值。

下面是简单 case 表达式的一个范例：

```
CASE customer.cust_type_cd
  WHEN 'I' THEN
   (SELECT CONCAT(i.fname, ' ', i.lname)
    FROM individual I
    WHERE i.cust_id = customer.cust_id)
  WHEN 'B' THEN
   (SELECT b.name
    FROM business b
    WHERE b.cust_id = customer.cust_id)
  ELSE 'Unknown Customer Type'
END
```

简单 case 表达式没有查找型 case 表达式强大，因为简单 case 表达式自动构建了等式条件，不是读者自己指定条件。为了解释这个问题，我写了一个和前面的简单 case 表达式逻辑相同的查找型 case 表达式:

```
CASE
  WHEN customer.cust_type_cd = 'I' THEN
   (SELECT CONCAT(i.fname, ' ', i.lname)
    FROM individual I
    WHERE i.cust_id = customer.cust_id)
  WHEN customer.cust_type_cd = 'B' THEN
   (SELECT b.name
    FROM business b
    WHERE b.cust_id = customer.cust_id)
  ELSE 'Unknown Customer Type'
END
```

利用查找型 case 表达式可以构建范围条件、不等条件以及基于 and/or/not 这些运算符的复合条件，所以我建议读者对所有非简单逻辑使用查找型 case 表达式。

11.3 case 表达式范例

下面提供一些范例阐明 SQL 语句中条件逻辑的用途。

11.3.1 结果集变换

读者可能遇到过这样的情况：对有限值集进行聚合时（比如对一周的天数聚合），希望结果集中每个值一个单列行而不是每个值一行。例如，要求读者写一个查询，展示从 2000 年到 2005 年每年的开户数目：

```
mysql> SELECT YEAR(open_date) year, COUNT(*) how_many
    -> FROM account
    -> WHERE open_date > '1999-12-31'
    ->   AND open_date < '2006-01-01'
    -> GROUP BY YEAR(open_date);
+------+----------+
| year | how_many |
+------+----------+
| 2000 |        3 |
| 2001 |        4 |
| 2002 |        5 |
| 2003 |        3 |
| 2004 |        9 |
+------+----------+
5 rows in set (0.00 sec)
```

如果要求读者返回一个单行 6 列的结果（每年一列）呢？为了将这个结果变换为单行，读者需要创建 6 列，并在每列中只对与所求年份相关的行求和：

```
mysql> SELECT
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2000 THEN 1
    ->         ELSE 0
    ->       END) year_2000,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2001 THEN 1
    ->         ELSE 0
    ->       END) year_2001,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2002 THEN 1
    ->         ELSE 0
    ->       END) year_2002,
```

```
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2003 THEN 1
    ->         ELSE 0
    ->       END) year_2003,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2004 THEN 1
    ->         ELSE 0
    ->       END) year_2004,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2005 THEN 1
    ->         ELSE 0
    ->       END) year_2005
    -> FROM account
    -> WHERE open_date > '1999-12-31' AND open_date < '2006-01-01';
+----------+----------+----------+---------+----------+---------+
| year_2000 | year_2001 | year_2002 | year_2003 | year_2004  | year_2005 |
+----------+----------+----------+---------+----------+---------+
|        3 |        4 |        5 |       3 |        9 |    0    |
+----------+----------+----------+---------+----------+---------+
1 row in set (0.01 sec)
```

除了年份值，前面查询中的 6 列都是相同的。 如果 extract() 函数返回的是该列需要的年份，那么 case 表达式返回 1，否则返回 0。显然，这种变换对小数目的值是可行的，但是如果统计要从 1905 年起，那么应用该变换将让人感到单调乏味。

提示

虽然对本书有点高级，但还是值得指出：SQL Server 和 Oracle 11g 为这些类型的查询特别包括了 PIVOT 子句。

11.3.2　选择性聚合

早在第 9 章中我展示了一个例子的部分解决方案，这个例子用于查找那些账户余额与 transaction 表中的原始数据不相符的账户。当时只提出了部分解决方案的原因是完全解决方案需要使用条件逻辑，因此现在解决这个问题的所有知识都已准备好了。下面是我在第 9 章中留下的问题：

```
SELECT CONCAT('ALERT! : Account #', a.account_id,
  ' Has Incorrect Balance!')
FROM account a
WHERE (a.avail_balance, a.pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
    SUM(<expression to generate pending balance>)
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

查询使用关联子查询对 transaction 表统计指定账户的个人交易。统计交易时，读者需考虑下面两个问题：

- 由于交易账户总是正的，所以读者需要查看交易类型是借款还是存款，借款则应该翻转标志（乘以-1）；
- 如果 funds_avail_date 列中的日期大于当前日期，交易应该被加到待收余额总和，而不是加到可用余额总和。

有些交易需要被排除在可用余额之外，而所有交易都应该被包含在待收余额之内，这就使后者成为两个计算中较简单的一个。下面的 case 表达式计算待收余额：

```
CASE
  WHEN transaction.txn_type_cd = 'DBT'
    THEN transaction.amount * -1
  ELSE transaction.amount
END
```

因此，所有账户计算借款交易时需要乘以-1，存款交易则不必。可用余额计算时也应用同样的逻辑，不过只有那些可用的交易才能被包含在内。因此，计算可用余额的 case 表达式包含一个额外的 when 子句：

```
CASE
  WHEN transaction.funds_avail_date > CURRENT_TIMESTAMP()
    THEN 0
  WHEN transaction.txn_type_cd = 'DBT'
    THEN transaction.amount * -1
  ELSE transaction.amount
END
```

有了第一个 when 子句的存在，那些不可用的资金（比如没有兑现的支票）对于总和贡献为$0。下面是使用了两个 case 表达式的最终查询：

```
SELECT CONCAT('ALERT! : Account #', a.account_id,
  ' Has Incorrect Balance!')
FROM account a
WHERE (a.avail_balance, a.pending_balance) <>
 (SELECT
    SUM(CASE
          WHEN t.funds_avail_date > CURRENT_TIMESTAMP()
            THEN 0
          WHEN t.txn_type_cd = 'DBT'
            THEN t.amount * -1
          ELSE t.amount
        END),
    SUM(CASE
          WHEN t.txn_type_cd = 'DBT'
```

```
            THEN t.amount * -1
          ELSE t.amount
        END)
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

通过使用条件逻辑，sum() 聚合函数接受两个 case 表达式处理过的数据，从而可以对恰当的交易额求和。

11.3.3　存在性检查

有时读者只希望确定两个实体之间是否存在某种关系而并不关心数量多少。比如，读者可能想知道某个客户是否有支票账户或者储蓄账户，但是并不关心每个类型的账户是否多于一个。下面的查询使用多个 case 表达式生成两个输出列，一列显示客户是否有支票账户，另一列显示他是否有储蓄账户：

```
mysql> SELECT c.cust_id, c.fed_id, c.cust_type_cd,
    ->   CASE
    ->     WHEN EXISTS (SELECT 1 FROM account a
    ->       WHERE a.cust_id = c.cust_id
    ->         AND a.product_cd = 'CHK') THEN 'Y'
    ->     ELSE 'N'
    ->   END has_checking,
    ->   CASE
    ->     WHEN EXISTS (SELECT 1 FROM account a
    ->       WHERE a.cust_id = c.cust_id
    ->         AND a.product_cd = 'SAV') THEN 'Y'
    ->     ELSE 'N'
    ->   END has_savings
    -> FROM customer c;
+---------+-------------+--------------+--------------+-------------+
| cust_id | fed_id      | cust_type_cd | has_checking | has_savings |
+---------+-------------+--------------+--------------+-------------+
|       1 | 111-11-1111 | I            | Y            | Y           |
|       2 | 222-22-2222 | I            | Y            | Y           |
|       3 | 333-33-3333 | I            | Y            | N           |
|       4 | 444-44-4444 | I            | Y            | Y           |
|       5 | 555-55-5555 | I            | Y            | N           |
|       6 | 666-66-6666 | I            | Y            | N           |
|       7 | 777-77-7777 | I            | N            | N           |
|       8 | 888-88-8888 | I            | Y            | Y           |
|       9 | 999-99-9999 | I            | Y            | N           |
|      10 | 04-1111111  | B            | Y            | N           |
|      11 | 04-2222222  | B            | N            | N           |
|      12 | 04-3333333  | B            | Y            | N           |
```

```
|      13 | 04-4444444  | B              | N              | N             |
+---------+-------------+----------------+----------------+---------------+
13 rows in set (0.00 sec)
```

每个 case 表达式包含了一个对 account 表的关联子查询：一个查找支票账户，另一个查找储蓄账户。由于每个 when 子句都使用了 exists 运算符，因此只要客户有至少一个所需的账户，那么条件就为真。

在其他情况下，读者可能关心涉及多少行，不过也只是在一定程度上。比如，下面的查询使用简单 case 表达式为每个客户计算账户数目，然后返回 None、1、2、3+：

```
mysql> SELECT c.cust_id, c.fed_id, c.cust_type_cd,
    ->  CASE (SELECT COUNT(*) FROM account a
    ->      WHERE a.cust_id = c.cust_id)
    ->    WHEN 0 THEN 'None'
    ->    WHEN 1 THEN '1'
    ->    WHEN 2 THEN '2'
    ->    ELSE '3+'
    ->  END num_accounts
    -> FROM customer c;
+---------+-------------+--------------+--------------+
| cust_id | fed_id      | cust_type_cd |num_accounts  |
+---------+-------------+--------------+--------------+
|       1 | 111-11-1111 | I            | 3+           |
|       2 | 222-22-2222 | I            | 2            |
|       3 | 333-33-3333 | I            | 2            |
|       4 | 444-44-4444 | I            | 3+           |
|       5 | 555-55-5555 | I            | 1            |
|       6 | 666-66-6666 | I            | 2            |
|       7 | 777-77-7777 | I            | 1            |
|       8 | 888-88-8888 | I            | 2            |
|       9 | 999-99-9999 | I            | 3+           |
|      10 | 04-1111111  | B            | 2            |
|      11 | 04-2222222  | B            | 1            |
|      12 | 04-3333333  | B            | 1            |
|      13 | 04-4444444  | B            | 1            |
+---------+-------------+--------------+--------------+
13 rows in set (0.01 sec)
```

在上面的查询中，我不想区分那些多于两个账户的客户，所以 case 表达式只是简单地创建了 3+类。或许，这个查询对于检索那些正联系银行打算开新账户的顾客是有用的。

11.3.4 除零错误

在执行包括除法的运算时，读者应该总是注意确保分母永远不能为 0。有些数据库服务器（比如 Oracle）在遇到 0 分母时将抛出一个错误，而 MySQL 只是简单地将结果值置

为 null，如下所示。

```
mysql> SELECT 100 / 0;
+---------+
| 100 / 0 |
+---------+
|    NULL |
+---------+
1 row in set (0.00 sec)
```

为了保证计算不遇到错误或者其他更糟糕的情况，也不会被莫名其妙地置为 null 值，读者应该将所有分母包装在条件逻辑里，如下所示。

```
mysql> SELECT a.cust_id, a.product_cd, a.avail_balance /
    ->   CASE
    ->     WHEN prod_tots.tot_balance = 0 THEN 1
    ->     ELSE prod_tots.tot_balance
    ->   END percent_of_total
    -> FROM account a INNER JOIN
    ->  (SELECT a.product_cd, SUM(a.avail_balance) tot_balance
    ->   FROM account a
    ->   GROUP BY a.product_cd) prod_tots
    ->   ON a.product_cd = prod_tots.product_cd;
+---------+------------+------------------+
| cust_id | product_cd | percent_of_total |
+---------+------------+------------------+
|      10 | BUS        |         0.000000 |
|      11 | BUS        |         1.000000 |
|       1 | CD         |         0.153846 |
|       6 | CD         |         0.512821 |
|       7 | CD         |         0.256410 |
|       9 | CD         |         0.076923 |
|       1 | CHK        |         0.014488 |
|       2 | CHK        |         0.030928 |
|       3 | CHK        |         0.014488 |
|       4 | CHK        |         0.007316 |
|       5 | CHK        |         0.030654 |
|       6 | CHK        |         0.001676 |
|       8 | CHK        |         0.047764 |
|       9 | CHK        |         0.001721 |
|      10 | CHK        |         0.322911 |
|      12 | CHK        |         0.528052 |
|       3 | MM         |         0.129802 |
|       4 | MM         |         0.321915 |
|       9 | MM         |         0.548282 |
|       1 | SAV        |         0.269431 |
|       2 | SAV        |         0.107773 |
|       4 | SAV        |         0.413723 |
|       8 | SAV        |         0.209073 |
|      13 | SBL        |         1.000000 |
```

```
+---------+-----------+-----------------+
24 rows in set (0.13 sec)
```

这个查询计算同一产品类型的所有账户的每个账户余额与总余额的比率。由于一些产品类型特殊，比如企业贷款，假设所有贷款现在都已被付清，余额总和可能是 0，因此，最好包括 case 表达式来确保分母永远不是 0。

11.3.5 有条件更新

读者在更新表中的行时，常常需要决定指定的列应该置什么值。比如，插入一个新的交易后，读者需要修改 account 表中 avail_balance、pending_balance 和 last_activity_date 这 3 列的值。虽然后两列比较容易更新，但是为了正确地修改 avail_balance 列，读者必须通过检查 transaction 表的 funds_avail_date 列判断交易资金是否立即可用。假定插入了 ID 为 999 的一个交易，读者可以用下面的 update 语句修改 account 表中的 3 列：

```
1   UPDATE account
2     SET last_activity_date = CURRENT_TIMESTAMP(),
3     pending_balance = pending_balance +
4      (SELECT t.amount *
5         CASE t.txn_type_cd WHEN 'DBT' THEN -1 ELSE 1 END
6       FROM transaction t
7       WHERE t.txn_id = 999),
8     avail_balance = avail_balance +
9      (SELECT
10         CASE
11           WHEN t.funds_avail_date > CURRENT_TIMESTAMP() THEN 0
12           ELSE t.amount *
13             CASE t.txn_type_cd WHEN 'DBT' THEN -1 ELSE 1 END
14         END
15        FROM transaction t
16        WHERE t.txn_id = 999)
17    WHERE account.account_id =
18     (SELECT t.account_id
19      FROM transaction t
20      WHERE t.txn_id = 999);
```

这个语句总共包含 3 个 case 语句：其中两个（第 5 行和第 13 行）对交易账户的借款额使用翻转符号，另一个 case 表达式（第 10 行）用于检查资金的可用性日期。如果日期是未来，则只对可用余额加 0，否则，应该加上这个交易额。

11.3.6 null 值处理

null 是在某列的值未知时存储到表中的值，不过检索时显示 null 值或者 null 值参与表达式这些情形并不总是合适的。例如，读者可能希望在数据输入屏幕上显示单词 unknown，而不是仅仅留下一个空白区域。检索数据时，读者可以在值为 null 时使用 case 表达式替换这个字符串，具体如下：

```
SELECT emp_id, fname, lname,
  CASE
    WHEN title IS NULL THEN 'Unknown'
    ELSE title
  END
FROM employee;
```

在计算中，null 值通常会导致一个 null 结果，如下所示。

```
mysql> SELECT (7 * 5) / ((3 + 14) * null);
+----------------------------+
| (7 * 5) / ((3 + 14) * null) |
+----------------------------+
|                       NULL |
+----------------------------+
1 row in set (0.08 sec)
```

执行运算时，case 表达式将 null 值转换为一个数字（通常是 0 或 1），从而使运算得到非 null 值结果。例如，假定计算包括 account.avail_balance 列，对于那些刚刚建立但尚未存款的账户，就可以用 0 替代（如果是加法或者减法）或者用 1 替代（如果是乘法或者除法）。

```
SELECT <some calculation> +
  CASE
    WHEN avail_balance IS NULL THEN 0
    ELSE avail_balance
  END
  + <rest of calculation>
...
```

如果一个数字列允许包含 null 值，那么在任何包含此列的计算中，为确保可以产生有用的结果，使用条件逻辑通常是一个好主意。

11.4　小测验

下面的例子考察读者解决条件逻辑问题的能力。完成练习后，请参照附录 C 检查答案。

练习 11-1

重写下面的查询，要求使用查找型 case 表达式替换简单 case 表达式，并且查询结果相同。请读者尽可能少使用 when 子句。

```
SELECT emp_id,
  CASE title
    WHEN 'President' THEN 'Management'
    WHEN 'Vice President' THEN 'Management'
    WHEN 'Treasurer' THEN 'Management'
    WHEN 'Loan Manager' THEN 'Management'
    WHEN 'Operations Manager' THEN 'Operations'
```

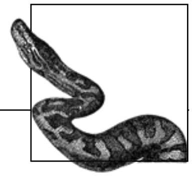
```
    WHEN 'Head Teller' THEN 'Operations'
    WHEN 'Teller' THEN 'Operations'
    ELSE 'Unknown'
  END
FROM employee;
```

练习 11-2

重写下面的查询,要求结果集为单行4列(每个分行1列)的,其中4列分别以branch_1～branch_4命名。

```
mysql> SELECT open_branch_id, COUNT(*)
    -> FROM account
    -> GROUP BY open_branch_id;
+----------------+----------+
| open_branch_id | COUNT(*) |
+----------------+----------+
|              1 |        8 |
|              2 |        7 |
|              3 |        3 |
|              4 |        6 |
+----------------+----------+
4 rows in set (0.00 sec)
```

第 12 章



事务

至此，本书中的所有范例都是一些单个的、独立的 SQL 语句。这可能是临时性报表或者数据维护脚本的典型形式，但应用程序逻辑通常包括多个 SQL 语句，它们作为一个逻辑工作单元一起执行。本章探讨多个 SQL 语句同时执行的必要性和所需的基础设施。

12.1 多用户数据库

数据库管理系统不但允许单个用户查询或者修改数据，而且可以多人同时操作。如果每个人都只执行查询，比如正常工作时间中数据仓库这种情况，那么数据库服务器就只有很少的问题要处理。然而，如果一些用户正在添加或者修改数据，那么服务器就必须做更多的簿记。

例如，读者正在生成一个报表，显示本行开立的所有支票账户的可用余额。但是，在生成报表的同时，发生了下面的活动：

- 一名收付员正在处理客户的存款；
- 一位客户正在前厅的 ATM 机上取钱；
- 银行的月结系统正在向账户支付利息。

因此，当读者生成报表时，还有多用户正在修改潜在的数据，报表上到底应该显示什么数据呢？答案在一定程度上取决于数据库如何处理锁。锁的有关内容将在下面介绍。

12.1.1 锁

锁是数据库服务器用来控制数据资源被并行使用的一种机制。当数据库的一些内容被锁定时，任何打算修改（或者可能是读取）这个数据的用户必须等到锁被释放。大部分数据库使用下面两种锁策略之一。

- 数据库的写操作必须向服务器申请并获得写锁才能修改数据，而读操作必须申请和获得读锁才能查询数据。多用户可以同时读取数据，而一个表（或其他部分）一次只能分配一个写锁，并且拒绝读请求直至写锁释放。
- 数据库的写操作必须向服务器申请并获得写锁才能修改数据，而读操作不需要任何类型的锁就可以查询数据。另一方面，服务器要保证从查询开始到结束读操作看到一个一致的数据视图（即使其他用户修改，数据看上去也要相同）。这个方法被称为版本控制。

这两种方法各有利弊。第一种方法在有较多的并行读请求和写请求时等待时间过长；如果在修改数据时存在长期运行的查询，则第二种方法也是有问题的。在本书讨论的 3 个服务器中，Microsoft SQL Server 采取第一种方法，Oracle 数据库使用第二种方法，MySQL 则两种方法都包括（取决于读者对存储引擎的选择，本章稍后讨论）。

12.1.2 锁的粒度

决定如何锁定一个资源时也可以采用一些不同的策略。服务器可能在 3 个不同级别之一应用锁，或者称作粒度：

表锁

阻止多用户同时修改同一个表的数据。

页锁

阻止多用户同时修改某表中同一页（一页通常是一段 2～16KB 的内存空间）的数据。

行锁

阻止多用户同时修改某表中同一行的数据。

同样，这些方法也是各有利弊。表锁需要较少的簿记就可以锁定整个表，但是用户增多时它会迅速产生不可接受的等待时间。另一方面，行锁需要更多的簿记，但是只要各个用户的兴趣在不同的行，它就能允许多人修改同一个表。本书讨论的 3 个数据库服务器中，Microsoft SQL Server 使用表锁、页锁和行锁，Oracle 数据库只有行锁，而 MySQL 采用表锁、页锁或行锁（同样取决于存取引擎的选择）。某些情况下，SQL Server 会将锁从行锁升级至页锁，再从页锁升级至表锁，然而 Oracle 数据库从不升级锁。

再回到上文谈及的报表问题，报表上的数据要么反映报表开始生成时的数据库状态（如果服务器使用版本控制方法），要么反映服务器为报表程序创建读锁时的数据库状态（如果服务器使用读锁和写锁）。

12.2 什么是事务

如果数据库正常运行时间为 100%、用户总是允许程序完成执行、应用程序总能完成而

不会遇到导致执行停止的错误，那么关于数据库并行存取就没有什么讨论的必要了。不过，我们可以断定上面这些状况不可能存在，因此允许多用户访问同一个数据还需要另外一个因素。

这个意外出现的并发难题是事务，它是一种将多条 SQL 语句聚集到一起，并且能够实现要么所有语句都执行，要么一个都不执行（这个属性称为原子性）。试想从储蓄账户划转$500 到支票账户时，如果钱已从储蓄账户成功支取却没有成功存入支票账户，那一定是件令人沮丧的事情。

为了避免这种错误，处理转账申请的程序将首先启动一个事务，然后发起 SQL 语句将钱从储蓄账户转到支票账户，如果所有事情成功，则发出 commit 命令结束事务；否则，如果有意外发生，就发出 rollback 命令撤销服务器自事务开始时的所有变化。整个过程大致如下：

```
START TRANSACTION;

 /* withdraw money from first account, making sure balance is sufficient */
UPDATE account SET avail_balance = avail_balance - 500
WHERE account_id = 9988
  AND avail_balance > 500;

IF <exactly one row was updated by the previous statement> THEN
  /* deposit money into second account */
  UPDATE account SET avail_balance = avail_balance + 500
    WHERE account_id = 9989;

  IF <exactly one row was updated by the previous statement> THEN
    /* everything worked, make the changes permanent */
    COMMIT;
  ELSE
    /* something went wrong, undo all changes in this transaction */
    ROLLBACK;
  END IF;
ELSE
  /* insufficient funds, or error encountered during update */
  ROLLBACK;
END IF;
```

提示

虽然前面的代码块看上去可能与大多数数据库公司提供的过程语言（比如 Oracle 的 PL、Microsoft 的 Transact-SQL）相似，但是这只是伪代码，也并没有想模仿哪种语言。

前面的代码块从启动事务开始，然后尝试从储蓄账户移走$500 并添加到支票账户。如果一切顺利，事务将被提交；如果出现了错误，事务将回滚，这意味着将撤销自事务

开始的所有数据变化。

通过使用事务，程序可以确保那$500 要么在储蓄账户，要么转移到支票账户，而不可能出什么意外。不管事务提交了还是回滚了，执行时获得的资源（比如写锁）在事务完成后都会被释放。

当然，如果程序设法完成两个 update 语句后，还没有执行 commit 或 rollback 命令，服务器突然宕机了，那么事务会在服务器重新上线后被回滚。（数据库服务器上线前必须完成的任务之一就是查找宕机前正在运行但未完成的事务，并将其回滚。）此外，如果程序完成了事务，并发出了 commit 指令，还没有将变化持久化到永久存储区（也就是说，修改的数据还位于内存，但没有被刷新到磁盘），服务器就宕机了，那么服务器重启时数据库服务器必须重新应用事务的变化（这种属性称为持久性）。

12.2.1 启动事务

数据库服务器以下面两种方法之一创建事务。

- 一个活跃事务总是和数据库会话相联系，所以没有必要，也没有什么方法能够显式地启动一个事务。当前事务结束时，服务器自动为会话启动一个新的事务。
- 如果不显式地启动一个会话，单个的 SQL 语句会被独立于其他语句自动提交。启动一个事务之前需先提交一个命令。

本书提及的 3 种服务器中，Oracle 数据库使用了第一种方法，Microsoft SQL Server 和 MySQL 则采取了第二种方法。Oracle 数据库事务创建方法的优点在于：即使在只提交单个 SQL 指令的情况下，如果读者不喜欢最终结果或者改变了主意，那么也有能力回滚所有的变化。

SQL:2003 标准包含了 start transaction 指令，它用来显式地启动一个事务。MySQL 遵守了标准，但是 SQL Server 用户必须使用替代命令 begin transaction。对于这两个服务器，读者一直处于所谓的自动提交模式，直到显式地启动一个事务，这意味着单个语句会被服务器自动提交。因此，读者可以自己决定进入事务模式并提交启动事务命令或者只是简单地让服务器提交单个语句。

MySQL 和 SQL Server 都允许读者为单个会话关闭自动提交模式，在这种情况下，对于事务来说服务器就像 Oracle 数据库一样工作。读者可以提交下面的命令关闭 SQL Server 的自动提交模式：

```
SET IMPLICIT_TRANSACTIONS ON
```

MySQL 允许读者以下面的方式关闭自动提交模式：

```
SET AUTOCOMMIT=0
```

一旦离开了自动提交模式，所有的 SQL 命令都会发生在同一个事务的范围，并且必须显式地对事务进行提交或者回滚。

提示

每次登录时关闭自动提交模式，并养成在事务内运行 SQL 语句的习惯。即使没有其他好处，至少能让读者免去请数据库管理员重建被自己无意删除的数据的难堪。

12.2.2 结束事务

一旦事务启动,不管是通过 start transaction 命令显式地启动还是由数据库服务器隐式地启动，为了持久化数据变化都必须显式地结束事务。读者可以通过 commit 指令解决这个问题，该指令命令服务器将变化标记为永久性的，进而释放事务中使用的任何资源(也就是页锁或者行锁)。

如果打算撤销自事务启动时所发生的一切变化，读者必须提交 rollback 指令，它命令服务器将数据返回到处理前的状态。同样，会话使用的任何资源都会在 rollback 完成后被释放。

除了提交 commit 或 rollback 指令，结束事务还可以由其他情景触发，要么作为活动的间接结果，要么作为意外的结果：

- 服务器宕机，在这种情况下，服务器重启时事务将会被自动回滚；
- 提交一个 SQL 模式语句，比如 alter table，这将会引起当前事务提交和一个新事务启动；
- 提交另一个 start transaction 命令，将会引起前一个事务提交；
- 因为服务器检测到一个死锁并且确定当前事务就是罪魁祸首，那么服务器就会提前结束当前的事务。这种情况下，事务将会被回滚，同时释放错误消息。

在这 4 个情景中，第一个和第三个比较容易理解，其他两个值得进行一番讨论。针对第二个情景来说，数据库的更改，无论是增加一个新表或新索引还是删除某表中的一列，都不能被回滚。因此，如果事务正在进行，那么服务器将会先提交当前事务，然后执行 SQL 模式语句命令，最后为会话自动启动一个新的事务。服务器不会通知读者到底发生了什么，所以读者应该注意保护那些组成一个工作单元的语句不被服务器意外地分成多个事务。

第四个情景处理死锁检测。死锁发生在两个不同的事务同时等待同一个资源，而这个资源正被另一个事务拥有的时候。例如，事务 A 可能刚好更新完 account 表，现在正等待 transaction 表的写锁，然而事务 B 向 transaction 表插入了一行，现在正等待 account 表的写锁。如果两个事务刚好修改同一页或者同一行（取决于数据库服务器使用的锁粒度)，那么它们都必须永远等待对方完成并释放自己所需的资源。数据库服务器必须一直注意这些情况才能使吞吐量不会陷入停滞。当检测到死锁时，必须选择一个事务（任意选择或者根据某个标准）回滚，这样其他事务才能继续下去。大多数情况下，终

止的事务可以重启，如果没有再次遇到另一个死锁情况它将会成功。

不像前面讨论的第二个情景，此时数据库服务器会抛出一个错误，并通知读者事务由于死锁检测已经被回滚。比如，MySQL 会返回 error #1213，它附带如下消息：

```
Message: Deadlock found when trying to get lock; try restarting transaction
```

正如错误消息所建议的，重试由于死锁检测而被回滚的事务是一种合理的做法。不过，如果死锁变得太频繁，那么读者可能需要修改访问数据库的程序以减少死锁的可能（一个常用的策略是总是按顺序访问数据资源，比如总是在插入交易数据前修改账户数据）。

12.2.3　事务保存点

在某些情况下，读者可能会遇到一些问题需要回滚事务，但是并不想撤销所有做过的工作。此时，读者可以在事务内创建一个或多个保存点，这样就可以利用它们回滚到事务的特定位置而不必一路回滚到事务启动状态。

选择存储引擎

当使用 Oracle 数据库和 Microsoft SQL Server 时，数据库都有单独的一套代码负责低级别数据库操作，比如用主键值从表中检索一个特定行。不过，MySQL 数据库服务器被设计成可以用多个存储引擎提供低级别的数据库功能，比如资源锁定和事务管理。6.0 版的 MySQL 包括以下存储引擎:

MyISAM

一种采用表级锁定的非事务引擎;

MEMORY

一种内存表使用的非事务引擎;

BDB

一种采用页级锁定的事务引擎;

InnoDB

一种采用行级锁定的事务引擎;

Merge

一种使多个相同的 MyISAM 看起来像一个单表（也叫表分割）的专用引擎;

Maria

6.0.6 版本中 MyISAM 的替代品，它添加了充分的恢复功能;

Falcon

6.0.4 版本起引入的采用行级锁定的高性能事务引擎;

Archive

一种用于存储大量未索引数据的专用引擎，主要用来存档。

读者可能认为自己不得不为数据库选择单一的数据引擎，但 MySQL 允许读者灵活地逐表选择一个存储引擎。对于那些可能参与事务的表，读者可以采用 InnoDB 或者 Falcon 存储引擎，它们使用行级锁定和版本控制提供所有存储引擎中最高级别的并行能力。

读者可以在创建表时显式地指定一个存储引擎，或者改变一个表的存储引擎。如果不知道表使用了什么引擎，那么读者可以使用 show table 命令，具体情况如下:

```
mysql> SHOW TABLE STATUS LIKE 'transaction' \G
*************************** 1. row ***************************
           Name: transaction
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 21
 Avg_row_length: 780
    Data_length: 16384
Max_data_length: 0
   Index_length: 49152
      Data_free: 0
 Auto_increment: 22
    Create_time: 2008-02-19 23:24:36
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
        Comment:
1 row in set (1.46 sec)
```

注意命令返回的第二项，读者会发现 transaction 表已经使用 InnoDB 引擎。如果没有指定引擎，读者可以通过下面的命令为 transaction 表指定 InnoDB 引擎:

```
ALTER TABLE transaction ENGINE = INNODB;
```

所有的保存点必须拥有一个名字，这样读者就可以在单个事务中拥有多个保存点。读者可以如下创建一个名为 my_savepoint 的保存点：

```
SAVEPOINT my_savepoint;
```

为了回滚到一个特定的保存点，读者只需简单地发出 rollback 命令，其后需跟关键词 to

savepoint 和保存点的名字，例如：

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

下面的例子展示如何使用保存点：

```
START TRANSACTION;

UPDATE product
SET date_retired = CURRENT_TIMESTAMP()
WHERE product_cd = 'XYZ';

SAVEPOINT before_close_accounts;

UPDATE account

SET status = 'CLOSED', close_date = CURRENT_TIMESTAMP(),
  last_activity_date = CURRENT_TIMESTAMP()
WHERE product_cd = 'XYZ';

ROLLBACK TO SAVEPOINT before_close_accounts;
COMMIT;
```

这个事务的影响是那个虚构的 XYZ 产品退出市场了，但涉及的账户并没有被关闭。

使用保存点时，读者需要记住下面两点。

- 创建保存点时，除了名字，什么都没有保存。为保证事务的持久化，读者最终必须发出一个 commit 命令。
- 如果读者发出一个没有保存点的 rollback 命令，那么事务中的所有保存点将被忽略，并且撤销整个事务。

如果使用 SQL Server，读者需使用专有命令 savetransaction 创建一个保存点，使用 rollback transaction 命令回滚一个保存点，这两个指令后面都需要跟保存点的名字。

12.3　小测验

下面的练习测试读者对事务的理解。完成习题后，请参照附录 C 检查答案。

练习 12-1

生成一个事务，它从 Frank Tucker 的货币市场账户存款转账$50 到他的支票账户。要求插入两行到 transaction 并更新 account 表中相应的两行内容。

第 13 章

索引和约束

由于本书关注于编程技术，所以前面 12 章集中讨论 SQL 语言的基础知识，利用前面所学读者可以精心构造一些强大的 select、insert、update 和 delete 语句。不过，数据库的其他功能也会间接影响读者所写的代码。本章重点探讨其中的两个功能：索引和约束。

13.1 索引

当向一个表中插入一行时，数据库服务器不会试图将数据放到表里任何特定的地方。例如，要向 department 表增加一行，那么服务器不会依据 dept_id 列的数字顺序或者 name 列的字母顺序存放该行。相反，服务器只是简单地将数据存放在文件中下一个可存放的位置（服务器为每个表预留了一系列空间）。因此，当查询 department 表时，服务器需要通过检查表中的每一行来完成查询。例如，对于下面的查询：

```
mysql> SELECT dept_id, name
    -> FROM department
    -> WHERE name LIKE 'A%';
+---------+----------------+
| dept_id | name           |
+---------+----------------+
|       3 | Administration |
+---------+----------------+
1 row in set (0.03 sec)
```

为了寻找所有名字以 A 开头的部门，服务器必须访问 department 表中的每一行并检查 name 列的内容。如果部门名以 A 开头，就将该行加入结果集。这种类型的访问称为表扫描。

这种方法对于只有 3 行的表来说效果不错，但是想象一下，如果一个表有 3 000 000 行，那么需要多长时间才能完成一次查询。对于大于 3 而小于 3 000 000 的数目，又遇到另

一个问题，就是如果没有其他帮助，服务器依然无法在合适的时间内完成查询。这个帮助就可以是 department 表中的一个或多个索引。

读者即使从来没有听说过数据库索引，也一定知道什么是索引（比如本书就有一个）。索引是寻找资源中特定项目的一种机制。例如，每个科技出版物结尾都有一个索引供读者定位其中的特定单词或者短语。索引依字母顺序列出这些单词或者短语，使读者能够快速定位到索引里的特定字母，找到所需条目，然后找到指定页或者单词或短语可能存在的那些页。

如同人们使用索引在出版物中查找单词一样，数据库服务器也使用索引定位表中的行。与普通的数据表不同，索引是一种以特定顺序保存的专用表。不过，索引并不包含实体中的所有数据，而是那些用于定位表中行的列，以及描述这些行的物理位置的信息。因此，索引的作用就是便捷化检索表中行和列的子集，而不需要检查表中的每行。

13.1.1 创建索引

再回到 department 表，为 name 列添加索引可以加速任何指定全部或者部分部门名字的查询以及 update 或 delete 操作。下面展示如何在 MySQL 数据库中添加这个索引：

```
mysql> ALTER TABLE department
    -> ADD INDEX dept_name_idx (name);
Query OK, 3 rows affected (0.08 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

这个语句为 department.name 列创建了索引（确切地说，这是一个 B 树，稍后再讨论），此外该索引被命名为 dept_name_idx。有了索引后，如果索引有利于改善查询，查询优化器（第 3 章中讨论过）就可以选择索引（假如 department 表中只有 3 行，那么优化器选择忽略索引而直接读取整个表可能会更合理）。如果表中的索引不止一个，那么优化器就必须判断对于特定的 SQL 语句使用哪个索引最有利。

提示

MySQL 将索引看做表的可选部件，所以读者必须使用 alter table 命令添加或者删除索引。其他数据库（包括 SQL Server 和 Oracle 数据库）则将索引视为独立的模式对象。对于 SQL Server 和 Oracle 数据库，读者可以使用 create index 命令生成索引，具体情况如下：

```
CREATE INDEX dept_name_idx
ON department (name);
```

5.0 版本的 MySQL 中 create index 命令已经映射到 alter table 命令，但前者仍然可用。

所有数据库服务器都允许读者查看可用索引。MySQL 用户可以使用 show 命令查看指定表中的所有索引，例如：

```
mysql> SHOW INDEX FROM department \G ************************ 1. row
************************* 1. row *************************
       Table: department
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: dept_id
    Collation: A
  Cardinality: 3
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_Comment:
************************* 2. row *************************
       Table: department
   Non_unique: 1
     Key_name: dept_name_idx
 Seq_in_index: 1
  Column_name: name
    Collation: A
  Cardinality: 3
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_Comment:
2 rows in set (0.01 sec)
```

结果显示 department 表共有两个索引：一个是 dept_id 列的索引 PRIMARY，另一个是 name 的索引 dept_name_idx。不过，到目前为止，我只创建了一个索引 dept_name_idx，那么另一个是从哪里来的呢？事实是这样的：当 department 表被创建时，create table 命令包含一个约束，这个约束将表中的 dept_id 列作为主关键字。下面的语句创建 department 表：

```
CREATE TABLE department
 (dept_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  name VARCHAR(20) NOT NULL,
  CONSTRAINT pk_department PRIMARY KEY (dept_id)  );
```

当表被创建时，MySQL 自动为主键列生成索引，在这个例子中主键列是 dept_id，生成索引名为 PRIMARY。约束将在本章稍后介绍。

创建索引后，如果读者觉得某个索引没用，就可以通过如下方法删除：

```
mysql> ALTER TABLE department
    -> DROP INDEX dept_name_idx;
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

提示

SQL Server 和 Oracle 数据库用户必须使用 drop index 命令删除索引：

```
DROP INDEX dept_name_idx; (Oracle)

DROP INDEX dept_name_idx ON department (SQL Server)
```

现在 MySQL 也支持 drop index 命令。

唯一索引

设计数据库时，考虑好哪些列能包含重复数据，哪些列不能是一件很重要的事情。例如，individual 表有两个名为 John Smith 的客户是允许的，因为每行有不同的标识符（cust_id）、出生日期和税务号码（customer.fed_id）可以帮助区分它们。但是，department 表中不能存在两个相同名字的部门。读者可以通过 department.name 创建唯一索引限制出现重复部门名字。

这里的唯一索引起了多重作用，除了提供常规索引的所有好处，还作为一种机制限制索引列出现重复值。无论是插入一行还是修改索引列，数据库服务器都会检查唯一索引以判断该值是否已存在于本表的某一行。下面显示了如何为 department.name 列创建唯一索引：

```
mysql> ALTER TABLE department
    -> ADD UNIQUE dept_name_idx (name);
Query OK, 3 rows affected (0.04 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

提示

SQL Server 和 Oracle 数据库的用户只需要在创建索引时增加关键字 unique，例如：

```
CREATE UNIQUE INDEX dept_name_idx
ON department (name);
```

有了合适的索引，如果读者试图添加一个名为 Operations 的部门，服务器就会抛出一个错误提示：

```
mysql> INSERT INTO department (dept_id, name)
    -> VALUES (999, 'Operations');
ERROR 1062 (23000): Duplicate entry 'Operations' for key 'dept_name_idx'
```

读者不必为主键列创建索引，因为服务器已经为主键检查唯一性。如果读者觉得有必要，还可以为同一个表创建不止一个唯一索引。

多列索引

除了上面涉及的单列索引，读者还可以创建跨越多列的索引。例如，使用姓氏和名字

查找雇员，读者就可以为这两列一起创建索引：

```
mysql> ALTER TABLE employee
    -> ADD INDEX emp_names_idx (lname, fname);
Query OK, 18 rows affected (0.10 sec)
Records: 18  Duplicates: 0  Warnings: 0
```

这个索引在两种查询中是有用的：一是指定了姓名，二是只指定了姓氏。但它不适合用于只指定客户名字的查询中，这是为什么？现在请读者考虑一下如何查找某个人的电话号码。如果知道此人的姓名就可以用电话簿快速查到号码，因为电话簿是先依据姓氏顺序，再依据名字顺序组织的；如果只知道此人名字就必须浏览电话簿中每个条目来查找具有指定名字的所有条目。

因此，在创建多列索引时，读者必须仔细考虑哪一列作为第一列，哪一列作为第二列等，这样索引才会尽可能的有用。请读者记住，如果需要保证充分的响应时间，也可以基于不同顺序为同一列集创建多列索引。

13.1.2　索引类型

索引是一种强大的工具，但是由于存在多种不同类型的数据，单一索引策略并不总是能满足需求。下面介绍各种服务器中采用的不同类型索引。

B 树索引

至此展示的所有索引都是平衡树索引，更为常用的称谓是 B 树索引。MySQL、Oracle 数据库和 SQL Server 默认都是 B 树索引，所以只要读者不显式地要求其他类型，索引就是 B 树索引。如读者所期望的，B 树索引以树结构组织，它有一个或多个分支节点，分支节点又指向单级的叶节点。分支节点用于遍历树，叶节点则保存真正的值和位置信息。例如，employee.lname 列的 B 树索引如图 13-1 所示。

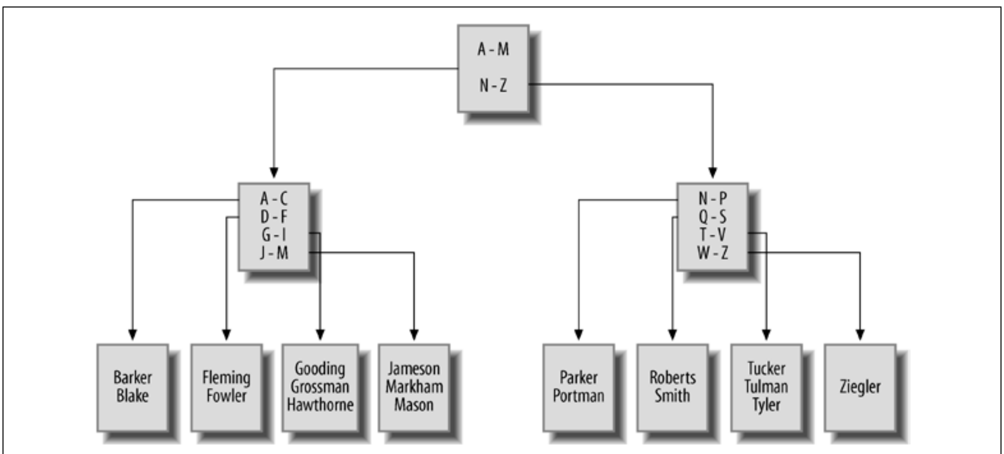


图 13-1　B 树示例

如果读者发起一个查询，检索所有姓氏以 G 开头的雇员，那么服务器将首先查找顶分支节点（称为根节点），接着顺着指针前进到姓氏以 A 到 M 开始的分支节点，然后服务器会在此分支节点内依次查看直至找到姓氏以 G 到 I 开始的叶节点，最后服务器开始读叶子中的数据直至遇到一个不以 G 开头的值（此时，这个值是 Hawthorne）。

当向 employee 表中插入、更新和删除行时，服务器会尽力保持树的平衡，这样就不会出现根节点的某一侧拥有比另一侧多得多的分支节点/叶节点。服务器通过增加或删除分支节点重新将值分配得更加均匀。通过保持树的均匀，不需要遍历多层分支节点，服务器就能够快速地到达叶节点查找到需要的值。

位图索引

虽然 B 树索引擅长于处理包含许多不同值的列，比如客户的姓氏/名字，但是在处理允许少量不同值的列时会变得很难使用。例如，为了能够快速检索某一特定类型（比如支票账户、储蓄账户）的所有账户，读者可能希望为 account.product_cd 列创建索引，但是，由于总共只有 8 种不同的产品，并且有些产品比其他产品受欢迎的多，所以客户数目的不断增长会使 B 树索引很难继续维持平衡。

对于那些包含少量值却占据了大量行（所谓低基数）的列，应该采用不同的索引策略。为了更有效地处理这个问题，Oracle 数据库引入了位图索引，它为存储在某列中的每个值生成一个位图。account.product_cd 中数据的位图索引大致如图 13-2 所示。

Value/row	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
BUS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
CD	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0
CHK	1	0	0	1	0	1	0	1	0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	0
MM	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
SAV	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
SBL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

图 13-2　位图示例

这个索引包含 6 个位图， 分别属于 product_cd 列中的每个值（8 个可用产品中有 2 个没被使用），并且每个位图为 account 表中的 24 行都分配了 0/1 值。因此，如果读者让服务器检索所有货币市场账户（product_cd = 'MM'），那么服务器只需简单地查找 MM 位图中的所有 1 值，最后返回第 7、10 和 18 行。如果读者查询多值，那么服务器也能联合位图。例如，假设要检索所有货币市场账户和存储账户（product_cd = 'MM' or product_cd = 'SAV'），则服务器将对 MM 和 SAV 位图执行或运算，最后返回第 2、5、7、9、10、16、18 行。

对于低基数而言，位图是一种友好、紧凑的索引解决方案，但是列中存储的值的数目攀升到相对行数太高时（所谓高基数），这种索引策略将会失败，因为服务器需要维护太多的位图。例如，读者永远不会为主键列创建一个位图索引，因为这代表最高可能的基数（每行都有不同的值）。

Oracle 用户可以通过为 create index 语句简单地添加关键词 bitmap 生成位图，例如：

```
CREATE BITMAP INDEX acc_prod_idx ON account (product_cd);
```

位图索引通常应用于数据仓库环境，那里会有大量数据被索引，那些列却只包含相对少的值（比如销售岗位、地理环境、产品、销售员）。

文本索引

如果数据库中存储文档，那么可能需要允许用户在文档中查找单词或者短语。我们当然不希望每次请求搜索时服务器都打开每个文档，然后扫描需要的文本，但是传统的索引策略又不适用于这种状况，那怎么办呢？为了处理这种状况，MySQL、SQL Server 和 Oracle 数据库为文档包含了专业的索引和搜索机制，其中前两者包含的是他们所称的全文索引（MySQL 中仅 MyISAM 存储引擎中可以使用全文索引），最后一个包含的是一种称为 Oracle Text 的强大工具集。文档查找非常专业，所以我就不举例子了，但还是希望读者至少能了解可用什么方法来处理。

13.1.3 如何使用索引

服务器通常首先利用索引快速定位特定表中的行，之后再访问相关表提取用户请求的补充信息。考虑下面的查询：

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE emp_id IN (1, 3, 9, 15);
+--------+---------+----------+
| emp_id | fname   | lname    |
+--------+---------+----------+
|      1 | Michael | Smith    |
|      3 | Robert  | Tyler    |
|      9 | Jane    | Grossman |
|     15 | Frank   | Portman  |
+--------+---------+----------+
4 rows in set (0.00 sec)
```

就本查询而言，服务器先使用 employee 表中 emp_id 列的主键索引定位 ID 为 1、3、9 和 15 的雇员，然后访问这 4 行，检索姓氏和名字两列。

不过，如果索引包含满足查询的所有内容，那么服务器就不必访问相关表了。为理解此点，请看看查询优化器如何使用不同的索引处理相同的查询。

下面的查询为指定的客户聚合账户余额：

```
mysql> SELECT cust_id, SUM(avail_balance) tot_bal
    -> FROM account
    -> WHERE cust_id IN (1, 5, 9, 11)
    -> GROUP BY cust_id;
+---------+----------+
| cust_id | tot_bal  |
+---------+----------+
|       1 |  4557.75 |
|       5 |  2237.97 |
|       9 | 10971.22 |
|      11 |  9345.55 |
+---------+----------+
4 rows in set (0.00 sec)
```

为了解 MySQL 查询优化器决定如何执行查询的，我使用了 explain 语句请求服务器显示查询的执行计划而不执行查询：

```
mysql> EXPLAIN SELECT cust_id, SUM(avail_balance) tot_bal
    -> FROM account
    -> WHERE cust_id IN (1, 5, 9, 11)
    -> GROUP BY cust_id \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: account
         type: index
possible_keys: fk_a_cust_id
          key: fk_a_cust_id
      key_len: 4
          ref: NULL
         rows: 24
        Extra: Using where
1 row in set (0.00 sec)
```

提示

每个数据库服务器都有工具供读者查看查询优化器是如何处理 SQL 语句的。SQL Serve 用户在要查询的 SQL 语句前加上 set showplan_text on 就可以查看该语句的执行计划。Oracle 数据库包含 explain plan 语句，它将执行计划写到一个专用表 plan_table 里。

下面就是执行计划告诉读者的内容，并未考虑太多细节：

- 使用索引 fk_a_cust_id 查找 account 表中满足 where 子句的行；
- 读取索引后，服务器预计会读取 account 表的所有 24 行以聚合可用余额数据，这是因为它不知道除了 ID 号为 1、5、9 和 11 的客户外，还可能有其他客户。

索引 fk_a_cust_id 是由服务器自动生成的又一个索引，但是这次自动生成的原因是外键约束而不是主键约束（本章稍后作更多介绍）。fk_a_cust_id 是 account.cust_id 列的索引，

因此服务器先使用索引定位 account 表中的 ID 为 1、5、9 和 11 的客户，然后访问这些行，再实现检索和聚合可用余额数据。

接下来，我给 cust_id 和 avail_balance 两列添加新索引 acc_bal_idx：

```
mysql> ALTER TABLE account
    -> ADD INDEX acc_bal_idx (cust_id, avail_balance);
Query OK, 24 rows affected (0.03 sec)
Records: 24  Duplicates: 0  Warnings: 0
```

下面看看有了这个索引后查询优化器是如何处理上一个查询的：

```
mysql> EXPLAIN SELECT cust_id, SUM(avail_balance) tot_bal
    -> FROM account
    -> WHERE cust_id IN (1, 5, 9, 11)
    -> GROUP BY cust_id \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: account
         type: range
possible_keys: acc_bal_idx
          key: acc_bal_idx
      key_len: 4
          ref: NULL
         rows: 8
        Extra: Using where; Using index
1 row in set (0.01 sec)
```

比较前面两个执行计划，可以得到下列区别：

- 优化器使用了新索引 acc_bal_idx，而不是 fk_a_cust_id；
- 优化器预期只需要 8 行，而不是 24 行；
- 不需要 account 表即可满足查询结果（使用附加列的索引指定）。

因此，服务器可以使用索引定位关联表中的行，或者只要索引包含查询需要的所有列，服务器可以把索引当做表一样使用。

提示

上面和读者一起探讨的是一个查询优化的例子。优化涉及查看 SQL 语句和决定可用于服务器执行语句的资源。为了更有效地执行，读者可以修改 SQL 语句，或者调整数据库资源，或者两者都做。优化是一个非常深入的话题，我也愿意强烈鼓励读者阅读自己服务器的优化指南或者挑选一本好的优化书自己学习研究，这样读者就会了解服务器上所有不同的可用方法。

13.1.4 索引的不足

既然索引如此有用，那么为什么不索引一切呢？恰当地说，这样做不一定是一件好事，

理解这个结论的关键是要知道每个索引事实上都是一个表（一种特殊类型的表，但也是表）。因此，每次在对表添加或者删除行时，表中的所有索引必须被修改；当更新行时，受到影响的列的任何索引也必须被修改。因此，索引越多，服务器就需要做越多的工作来保持所有模式对象最新，当然，这将会拖慢服务器处理任务的速度。

索引需要磁盘空间，同时也需要管理员耗费一些精力去管理它们，因此对于索引的最佳策略是：仅当出现清晰需求时才添加索引。如果有特殊目的需要索引，比如每月例行维护程序，那么读者可以添加索引，运行程序，然后删除索引，下次需要时再如此重复一遍。对于数据仓库来说，用户在营业期间生成报表和特定查询时，索引至关重要，但是当数据被彻夜装载到数据仓库时，就会出现问题，所以一种常见的做法是：装载数据前删除索引，然后在仓库开放营业前重建它们。

一般来说，读者应该努力避免使用太多索引和太少索引。如果不能确定到底需要多少索引，读者可以将下面的内容作为默认策略使用。

- 确保所有主键列被索引（大部分服务器会在创建主键约束时自动生成唯一索引）。针对多列主键，考虑为主键列的子集构建附加索引，或者以与主键约束定义不同的顺序为所有主键列另外生成索引。
- 为所有被外键约束引用的列创建索引。服务器在准备删除父行时会查找以确保没有子行存在，为此它必须发出一个查询搜索列中的特定值，如果该列没有索引，那么服务器必须扫描整个表。
- 索引那些被频繁检索的列。除了短字符串（3～50 个字符）列，大多数日期列也是不错的候选。

读者在创建一套初始索引后，尽力捕获并分析对表的真实查询，然后修改索引策略以满足最常见的访问路径。

13.2 约束

约束是一种简单地强加于表中一列或多列的限制。约束有以下几种不同的类型。

主键约束

标志一列或多列，并保证其值在表内的唯一性。

外键约束

限制一列或多列中的值必须被包含在另一表的外键列中，并且在级联更新或级联删除规则建立后也可以限制其他表中的可用值。

唯一约束

限制一列或多列的值，保证其在表内的唯一性（主键约束是一种特殊类型的唯一约束）。

检查约束

限制一列的可用值范围。

没有了约束，一个数据库的一致性就会令人怀疑。例如，如果服务器允许只修改 customer 表中的客户 ID 而不改变 account 表中的同一个 ID，那么最终结果就是账户不再指向合法的客户记录（所谓孤儿行）。不过，有了合适的主键和外键约束后，服务器就可以在试图修改或删除被其他表引用的数据时要么抛出一个错误，要么将这些改变传播到其他表（稍后会作更多讨论）。

提示

如果读者希望在 MySQL 服务器中应用外键约束，那么表的存储引擎必须是 InnoDB。6.0.4 版本中 Falcon 引擎不支持这种约束，不过后来的版本中都实现了这种约束的支持。

13.2.1 创建约束

约束一般与关联表通过 create table 语句同时创建。为了说明这个问题，下面的例子引入了本书示例数据库的模式生成脚本：

```
CREATE TABLE product
 (product_cd VARCHAR(10) NOT NULL,
  name VARCHAR(50) NOT NULL,
  product_type_cd VARCHAR (10) NOT NULL,
  date_offered DATE,
  date_retired DATE,
      CONSTRAINT fk_product_type_cd FOREIGN KEY (product_type_cd)
        REFER ENCES product_type (product_type_cd),
      CONSTRAINT pk_product PRIMARY KEY (product_cd)
 );
```

Product 表包含两个约束：一个指定了 product_cd 列作为表的主键，另一个指定了 product_type_cd 列作为来自 product_type 表的外键。或者，读者可以先仅仅创建 product 表而不指定约束，然后通过 alter table 语句添加主键约束和外键约束：

```
ALTER TABLE product
ADD CONSTRAINT pk_product PRIMARY KEY (product_cd);

ALTER TABLE product
ADD CONSTRAINT fk_product_type_cd FOREIGN KEY (product_type_cd)
  REFERENCES product_type (product_type_cd);
```

如果希望删除主键约束或者外键约束，那么读者可以再使用 alter table 语句，不过这次需要使用 drop 代替 add：

```
ALTER TABLE product
DROP PRIMARY KEY;

ALTER TABLE product
DROP FOREIGN KEY fk_product_type_cd;
```

删除一个主键列并不常见，不过在某些维护操作中外键约束有时会被删除，随后再重建。

13.2.2 约束与索引

正如读者在本章前面所看到的，创建约束有时可能导致自动创建一个索引。不过，数据库服务器对于约束和索引的关系有着不同的处理原则。表 13-1 显示了 MySQL、SQL Server 和 Oracle 数据库是如何处理约束与索引之间的关系的。

表 13-1 约束生成

约束类型	MySQL	SQL Server	Oracle 数据库
主键约束	生成唯一索引	生成唯一索引	使用已存在的索引或创建新索引
外键约束	生成索引	不生成索引	不生成索引
唯一约束	生成唯一索引	生成唯一索引	使用已存在的索引或创建新索引

由此可见，在实施主键约束、外键约束和唯一约束时，MySQL 生成新索引；SQL Server 只为主键约束和唯一约束生成新索引，不管外键约束；Oracle 数据库除了使用已经存在的索引外（如果存在合适的索引），基本上采取与 SQL Server 相同的方法。虽然 SQL Server 和 Oracle 数据库都不为外键约束生成索引，但是它们的文档中建议为每个外键创建索引。

13.2.3 级联约束

有了合适的外键约束后，如果读者试图插入新行或者修改行而导致父表中的外键列并无可匹配值，那么服务器会抛出一个错误。为解释这个问题，下面看看 product 表和 product_type 表中的数据：

```
mysql> SELECT product_type_cd, name
    -> FROM product_type;
+-----------------+-------------------------------+
| product_type_cd | name                          |
+-----------------+-------------------------------+
| ACCOUNT         | Customer Accounts             |
| INSURANCE       | Insurance Offerings           |
| LOAN            | Individual and Business Loans |
+-----------------+-------------------------------+
3 rows in set (0.00 sec)

mysql> SELECT product_type_cd, product_cd, name
    -> FROM product
    -> ORDER BY product_type_cd;
+-----------------+------------+------------------------+
| product_type_cd | product_cd | name                   |
+-----------------+------------+------------------------+
| ACCOUNT         | CD         | certificate of deposit |
```

```
| ACCOUNT          | CHK        | checking account        |
| ACCOUNT          | MM         | money market account    |
| ACCOUNT          | SAV        | savings account         |
| LOAN             | AUT        | auto loan               |
| LOAN             | BUS        | business line of credit |
| LOAN             | MRT        | home mortgage           |
| LOAN             | SBL        | small business loan     |
+------------------+------------+-------------------------+
8 rows in set (0.01 sec)
```

product_type 表中的 product_type_cd 列有 3 个不同的值（ACCOUNT、INSURANCE 和 LOAN），其中的两个（ACCOUNT 和 LOAN）在 product 表中的 product_type_cd 列被引用。

下面的语句试图将 product 表中的 product_type_cd 列更改为 product_type 表中根本不存在的值：

```
mysql> UPDATE product
    -> SET product_type_cd = 'XYZ'
    -> WHERE product_type_cd = 'LOAN';
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint
fails ('bank'.'product', CONSTRAINT 'fk_product_type_cd' FOREIGN KEY
('product_type_cd') REFERENCES 'product_type' ('product_type_cd'))
```

product.product_type_cd 列上有外键约束，而 product_type 表中没有哪一行的 product_type_cd 列值为 XYZ，所以服务器的更新是不会成功的。也就是说，如果父行没有相应的值，外键约束不允许更改子行。

不过，如果试图更改 product_type 表中的父行为 XYZ，那又将如何呢？下面的更新语句试图将产品类型 LOAN 更改为 XYZ：

```
mysql> UPDATE product_type
    -> SET product_type_cd = 'XYZ'
    -> WHERE product_type_cd = 'LOAN';
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key
constraint fails ('bank'.'product', CONSTRAINT 'fk_product_type_cd'
FOREIGN KEY
('product_type_cd') REFERENCES 'product_type' ('product_type_cd'))
```

再次抛出一个错误，不过这次因为 product 表中存在子行的 product_type_cd 列值为 LOAN。这是外键约束的默认做法，不过这并不是唯一可能的做法，相反，读者可以命令服务器帮助自己将变化传播到所有子行，这样就保证了数据的完整性。所谓级联更新，在删除存在的外键和添加新的外键时包含 on update cascade 语句，这种外键约束的变化能够实现传播。

```
mysql> ALTER TABLE product
    -> DROP FOREIGN KEY fk_product_type_cd;
Query OK, 8 rows affected (0.02 sec)
```

```
Records: 8  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE product
    -> ADD CONSTRAINT fk_product_type_cd FOREIGN KEY (product_type_cd)
    ->  REFERENCES product_type (product_type_cd)
    ->  ON UPDATE CASCADE;
Query OK, 8 rows affected (0.03 sec)
Records: 8  Duplicates: 0  Warnings: 0
```

对约束进行了合适的修改后，再看看前面的 update 语句试图执行时会如何：

```
mysql> UPDATE product_type
    -> SET product_type_cd = 'XYZ'
    -> WHERE product_type_cd = 'LOAN';
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

这一次语句执行成功。为证明这些改变已经传播到 product 表，再看看两表的数据：

```
mysql> SELECT product_type_cd, name
    -> FROM product_type;
+-----------------+--------------------------------+
| product_type_cd | name                           |
+-----------------+--------------------------------+
| ACCOUNT         | Customer Accounts              |
| INSURANCE       | Insurance Offerings            |
| XYZ             | Individual and Business Loans |
+-----------------+--------------------------------+
3 rows in set (0.02 sec)

mysql> SELECT product_type_cd, product_cd, name
    -> FROM product
    -> ORDER BY product_type_cd;
+-----------------+------------+-------------------------+
| product_type_cd | product_cd | name                    |
+-----------------+------------+-------------------------+
| ACCOUNT         | CD         | certificate of deposit  |
| ACCOUNT         | CHK        | checking account        |
| ACCOUNT         | MM         | money market account    |
| ACCOUNT         | SAV        | savings account         |
| XYZ             | AUT        | auto loan               |
| XYZ             | BUS        | business line of credit |
| XYZ             | MRT        | home mortgage           |
| XYZ             | SBL        | small business loan     |
+-----------------+------------+-------------------------+
8 rows in set (0.01 sec)
```

正如读者所见，product_type 表的改变已经传播到了 product 表。除了级联更新，读者还可以指定级联删除。如果父表中的一行被删除，那么级联删除就会删除子表中的行。使用 on delete cascade 可以指定级联删除，例如：

```
ALTER TABLE product
ADD CONSTRAINT fk_product_type_cd FOREIGN KEY (product_type_cd)
```

```
    REFERENCES product_type (product_type_cd)
    ON UPDATE CASCADE
    ON DELETE CASCADE;
```

有了这个版本的约束，当 product_type 表中的一行被更新时，服务器将更新 product 表中的子行，同样，如果 product_type 表中的行被删除，product 表中的子行也将被删除。

级联约束是一个盒子，其中的约束直接影响读者写的代码。读者要知道自己服务器中哪个约束指定了级联更新/删除，这样才能预测到 update 和 delete 语句的所有影响。

13.3 小测验

下面的练习测试读者索引和约束的知识。完成练习题后，参考附录 C 检查答案。

练习 13-1

修改 account 表，使客户不能在任何产品中拥有多个账户（最多一个）。

练习 13-2

为 transaction 表生成多列索引，该索引可用于如下两个查询。

```
SELECT txn_date, account_id, txn_type_cd, amount
FROM transaction
WHERE txn_date > cast('2008-12-31 23:59:59' as datetime);

SELECT txn_date, account_id, txn_type_cd, amount
FROM transaction
WHERE txn_date > cast('2008-12-31 23:59:59' as datetime)
  AND amount < 1000;
```

第 14 章

视图

精心设计的程序通常会在不断完善私有细节时公开一个公共接口，这样未来可以在不影响终端用户的情况下修改设计。设计数据库时，读者可以采取保持表私有并只允许用户通过一系列视图访问数据的策略。利用这种策略，读者可以获取与程序接口相似的效果。本章致力于定义什么是视图，如何创建它们，何时以及如何使用它们。

14.1　什么是视图

视图是一种简单的数据查询机制。不同于表，视图不涉及数据存储，因此读者不用担心视图会充满磁盘空间。读者可以先通过命名 select 语句来创建视图，然后将这个查询保存起来供其他用户使用，而其他用户使用这个视图时就像他们自己在直接查询数据（事实上，他们可能不知道自己正在使用一个视图）。

举一个简单的例子，假设读者想部分掩盖 customer 表中的联邦个人识别号码（社会安全号和企业识别号）。例如，客户服务部门可能只需要访问联邦个人识别号码的最后一部分以验证电话访客的身份，因为公开所有数字将会违反公司的隐私政策。解决方案不是允许所有银行雇员直接访问 customer 表，而是先定义一个名为 customer_vw 的视图，然后授权他们使用它访问客户数据。下面是这个视图的定义：

```
CREATE VIEW customer_vw
 (cust_id,
  fed_id,
  cust_type_cd,
  address,
  city,
  state,
  zipcode
 )
```

```
AS
SELECT cust_id,
  concat('ends in ', substr(fed_id, 8, 4)) fed_id,
  cust_type_cd,
  address,
  city,
  state,
  postal_code
FROM customer;
```

上面语句的第一部分列出了视图的列名，它们可能与基础表的不同（比如，customer_vw 视图中 zipcode 列就映射到 customer.postal_code 列，但列名不同）；第二部分是一个 select 语句，它的作用是为视图中的每列提供一个表达式。

执行 create view 语句时，数据库服务器只是简单地存储视图的定义为将来使用。如果不执行查询就不会检索或存储任何数据。一旦视图被创建，用户就能把它当做一个表来查询，例如：

```
mysql> SELECT cust_id, fed_id, cust_type_cd
    -> FROM customer_vw;
+---------+-------------+--------------+
| cust_id | fed_id      | cust_type_cd |
+---------+-------------+--------------+
|       1 | ends in 1111 | I            |
|       2 | ends in 2222 | I            |
|       3 | ends in 3333 | I            |
|       4 | ends in 4444 | I            |
|       5 | ends in 5555 | I            |
|       6 | ends in 6666 | I            |
|       7 | ends in 7777 | I            |
|       8 | ends in 8888 | I            |
|       9 | ends in 9999 | I            |
|      10 | ends in 111  | B            |
|      11 | ends in 222  | B            |
|      12 | ends in 333  | B            |
|      13 | ends in 444  | B            |
+---------+-------------+--------------+
13 rows in set (0.02 sec)
```

服务器真正执行的查询不是用户提交的那个，也不是将提交的直接附加到视图定义而组成的查询，而是将两者合并创建的一个新查询。此例中的新查询如下：

```
SELECT cust_id,
  concat('ends in ', substr(fed_id, 8, 4)) fed_id,
  cust_type_cd
FROM customer;
```

即使视图 customer_vw 的定义中包含 customer 中的 7 列，服务器执行查询时也只是检索了其中的 3 个。正如你将在本章后面要看到的，如果视图中的一些列被附加到函数

或子查询，那么这将是一个重要的区别。

从用户的角度来看，视图看起来确实像一个表。如果想知道视图中有哪些可用列，读者可以使用 MySQL（或 Oracle）的 describe 命令查询：

```
mysql> describe customer_vw;
+--------------+------------------+------+-----+---------+-------+
| Field        | Type             | Null | Key | Default | Extra |
+--------------+------------------+------+-----+---------+-------+
| cust_id      | int(10) unsigned | NO   |     | 0       |       |
| fed_id       | varchar(12)      | YES  |     | NULL    |       |
| cust_type_cd | enum('I','B')    | NO   |     | NULL    |       |
| address      | varchar(30)      | YES  |     | NULL    |       |
| city         | varchar(20)      | YES  |     | NULL    |       |
| state        | varchar(20)      | YES  |     | NULL    |       |
| postal_code  | varchar(10)      | YES  |     | NULL    |       |
+--------------+------------------+------+-----+---------+-------+
7 rows in set (1.40 sec)
```

读者查询视图时，可以自由使用 select 语句中的任何子句，包括 group by、having 和 order by。下面就是一个例子：

```
mysql> SELECT cust_type_cd, count(*)
    -> FROM customer_vw
    -> WHERE state = 'MA'
    -> GROUP BY cust_type_cd
    -> ORDER BY 1;
+--------------+----------+
| cust_type_cd | count(*) |
+--------------+----------+
| I            |        7 |
| B            |        2 |
+--------------+----------+
2 rows in set (0.22 sec)
```

另外，读者也可以在查询里连接视图到其他表（甚至是视图），例如：

```
mysql> SELECT cst.cust_id, cst.fed_id, bus.name
    -> FROM customer_vw cst INNER JOIN business bus
    ->   ON cst.cust_id = bus.cust_id;
+---------+------------+------------------------+
| cust_id | fed_id     | name                   |
+---------+------------+------------------------+
|      10 | ends in 111 | Chilton Engineering    |
|      11 | ends in 222 | Northeast Cooling Inc. |
|      12 | ends in 333 | Superior Auto Body     |
|      13 | ends in 444 | AAA Insurance Inc.     |
+---------+------------+------------------------+
4 rows in set (0.24 sec)
```

为了只检索企业客户，这个查询将视图 customer_vw 连接到了 business 表。

14.2 为什么使用视图

在上一节中介绍的简单视图的唯一目的是掩盖 customer.fed_id 列的内容。视图通常被用于此类目的，但是还有更多的理由，下面将详细介绍。

14.2.1 数据安全

如果读者创建一个表并允许用户查询，那么他们将能够访问表中的每行每列。正如读者早就指出的，表里的一些列可能包含敏感数据，比如身份证号码或信用卡号码，把这些数据对所有用户公开不仅不是一个好主意，还可能违反公司的隐私规定，甚至是州或联邦法律。

这种状况下最好的办法就是首先保持表的私有权限（比如，不向用户授权 select 许可），然后创建一个或多个视图，这些视图省略或者掩盖了（比如处理 customer_vw.fed_id 列时采用以####结尾的方法）敏感内容。读者也可以在视图定义中添加 where 子句，限制用户只能访问那些被允许的行。例如，下面定义的视图只允许查询企业客户：

```
CREATE VIEW business_customer_vw
 (cust_id,
  fed_id,
  cust_type_cd,
  address,
  city,
  state,
  zipcode
 )
AS
SELECT cust_id,
  concat('ends in ', substr(fed_id, 8, 4)) fed_id,
  cust_type_cd,
  address,
  city,
  state,
  postal_code
FROM customer
WHERE cust_type_cd = 'B'
```

如果读者将这个视图提供给银行营业部，那么他们将只能够访问那些企业客户，这是因为查询中永远包含视图中的 where 子句条件。

提示

Oracle 数据库用户还有另一个选择可以保证表中的行与列的安全: 虚拟私有数据库（VPD）。VPD 允许读者对表施加策略，这样服务器为了执行策略就会在必要时修改用户的查询。例如，读者制定这样一个策略：公司银行营业部员工只能看到企业账户，然后就在每个对 customer 表的查询后添加条件 cust_type_cd = 'B'。

14.2.2 数据聚合

报表程序通常需要聚合数据，视图就是实现这一功能的好方法，使数据像已经被预聚合并存储在数据库一样。举例来说，假设需要一个应用程序每月都生成报表展示账户数目和每个客户的储蓄总额。读者可以为程序开发人员提供如下视图，而不是允许其直接查询基本表：

```
CREATE VIEW customer_totals_vw
 (cust_id,
  cust_type_cd,
  cust_name,
  num_accounts,
  tot_deposits
 )
AS
SELECT cst.cust_id, cst.cust_type_cd,
  CASE
    WHEN cst.cust_type_cd = 'B' THEN
     (SELECT bus.name FROM business bus WHERE bus.cust_id = cst.cust_id)
    ELSE
     (SELECT concat(ind.fname, ' ', ind.lname)
      FROM individual ind
      WHERE ind.cust_id = cst.cust_id)
  END cust_name,
  sum(CASE WHEN act.status = 'ACTIVE' THEN 1 ELSE 0 END) tot_active_accounts,
  sum(CASE WHEN act.status = 'ACTIVE' THEN act.avail_balance ELSE 0 END)
tot_balance
FROM customer cst INNER JOIN account act
  ON act.cust_id = cst.cust_id
GROUP BY cst.cust_id, cst.cust_type_cd;
```

使用这种方法为数据库设计者提供了很大的灵活性。如果将来某天读者需要大幅提高查询效率，决定将数据预聚合到一个表中而不是利用视图总计，那么读者可以先创建一个 customer_totals 表，然后修改视图 customer_totals_vw 的定义，再从该表中检索数据。在修改视图定义前，读者可以使用它来填充新表。下面是用于这个情景所需要的 SQL 语句：

```
mysql> CREATE TABLE customer_totals
    -> AS
    -> SELECT * FROM customer_totals_vw;
Query OK, 13 rows affected (3.33 sec)
Records: 13  Duplicates: 0  Warnings: 0

mysql> CREATE OR REPLACE VIEW customer_totals_vw
    ->  (cust_id,
    ->   cust_type_cd,
    ->   cust_name,
```

```
    ->   num_accounts,
    ->   tot_deposits
    ->  )
    -> AS
    -> SELECT cust_id, cust_type_cd, cust_name, num_accounts, tot_deposits
    -> FROM customer_totals;
Query OK, 0 rows affected (0.02 sec)
```

这样,现在的所有查询都用视图 customer_totals_vw 从新表 customer_totals 中提取数据,这意味着用户无需修改查询就能得到性能提升。

14.2.3 隐藏复杂性

部署视图最常见的理由之一是为终端用户屏蔽复杂性。例如，假设需要每月创建一个报表展示雇员数目、活跃账户总数和每个分行的交易总数。读者应该为报表设计者提供如下视图，而不应该期望他们能浏览 4 个不同的表：

```
CREATE VIEW branch_activity_vw
 (branch_name,
  city,
  state,
  num_employees,
  num_active_accounts,
  tot_transactions
 )
AS
SELECT br.name, br.city, br.state,
 (SELECT count(*)
  FROM employee emp
  WHERE emp.assigned_branch_id = br.branch_id) num_emps,
 (SELECT count(*)
  FROM account acnt
  WHERE acnt.status = 'ACTIVE' AND acnt.open_branch_id = br.branch_id)
num_accounts,
 (SELECT count(*)
  FROM transaction txn
  WHERE txn.execution_branch_id = br.branch_id) num_txns
FROM branch br;
```

这个报表定义很有趣，因为 6 列中有 3 列的值是使用标量子查询生成的。如果某个用户使用了这个视图却没有引用 num_employees、num_active_accounts 或 tot_transactions 列，那么一个子查询都不会执行。

14.2.4 连接分区数据

为了提升性能，一些数据库在设计时将大表分成多个小块。例如，transaction 变得越来越大，设计者就可以将其分为两个表：transaction_current，保存最近 6 个月的数据；transaction_historic，保存 6 个月前的所有数据。如果客户想查看特定账号的所有交易，

就需要查询两个表。不过，通过查询两表的视图和联合两个查询结果，同样可以使所有交易数据就像存储在一个单表中一样简单。这个视图的定义如下：

```
CREATE VIEW transaction_vw
 (txn_date,
  account_id,
  txn_type_cd,
  amount,
  teller_emp_id,
  execution_branch_id,
  funds_avail_date
 )
AS
SELECT txn_date,  account_id,  txn_type_cd,  amount,  teller_emp_id,
  execution_branch_id,  funds_avail_date
FROM transaction_historic
UNION ALL
SELECT txn_date,  account_id,  txn_type_cd,  amount,  teller_emp_id,
  execution_branch_id,  funds_avail_date
FROM transaction_current;
```

这种情况下使用视图是一个好主意，因为它允许设计者更改基础数据结构而不必强迫所有数据库用户修改它们的查询。

14.3　可更新的视图

如果提供给用户一系列视图作为检索数据使用，那么用户也需要修改同一数据怎么办？强迫用户使用视图检索，又允许他们使用 update 或 insert 语句自己修改基础数据。这似乎有点奇怪。为此，MySQL、Oracle 数据库和 SQL Server 都允许用户在遵守特定规则的前提下通过视图修改数据。对于 MySQL 来说，如果下面的条件能够满足，那么视图就是可更新的：

- 没有使用聚合函数（max()、min()和 avg()等）；
- 视图没有使用 group by 或 having 子句；
- select 或 from 子句中不存在子查询，并且 where 子句中的任何子查询都不引用 from 子句中的表；
- 视图没有使用 union、union all 和 distinct；
- from 子句包括不止一个表或可更新视图；
- 如果有不止一个表或视图，那么 from 子句只使用内连接。

为了说明可更新视图的用途，我们先从简单视图定义开始，然后逐步进入更加复杂的视图。

14.3.1　更新简单视图

本章开头的视图只是简单地读取数据，下面再看看这个例子：

```
CREATE VIEW customer_vw
 (cust_id,
  fed_id,
  cust_type_cd,
  address,
  city,
  state,
  zipcode
 )
AS
SELECT cust_id,
  concat('ends in ', substr(fed_id, 8, 4)) fed_id,
  cust_type_cd,
  address,
  city,
  state,
  postal_code
FROM customer;
```

视图 customer_vw 查询一个单表，并且通过一个表达式只导出 7 列中的 1 列。这个定义没有违反前面列出的任何限制，所以读者可以使用它修改 customer 表中的数据，例如：

```
mysql> UPDATE customer_vw
    -> SET city = 'Woooburn'
    -> WHERE city = 'Woburn';
Query OK, 1 row affected (0.34 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

正如读者所看到的，该语句已经修改了 1 行，下面可以通过检查基础表 customer 确认此结果：

```
mysql> SELECT DISTINCT city FROM customer;
+------------+
| city       |
+------------+
| Lynnfield  |
| Woooburn   |
| Quincy     |
| Waltham    |
| Salem      |
| Wilmington |
| Newton     |
+------------+
7 rows in set (0.12 sec)
```

由上面的查询结果可知，视图的修改成功了。显然，按照这种方式，读者能够修改视

图中的大多数列，但无法修改 fed_id 列，这是因为它是由一个表达式生成的：

```
mysql> UPDATE customer_vw
    -> SET city = 'Woburn', fed_id = '999999999'
    -> WHERE city = 'Woooburn';
ERROR 1348 (HY000): Column 'fed_id' is not updatable
```

这种情况可能并不是一件坏事，因为创建这个视图的所有目的只是掩盖联邦个人识别号码。

如果读者想用视图 customer_vw 插入数据，那就要失望了，因为包含导出列的视图不能用于插入数据，即使导出列没有被包含在插入语句中。例如，下面的语句试图用视图 customer_vw 向 cust_id、 cust_type_cd 和 city 列插入数据：

```
mysql> INSERT INTO customer_vw(cust_id, cust_type_cd, city)
    -> VALUES (9999, 'I', 'Worcester');
ERROR 1471 (HY000): The target table customer_vw of the INSERT is not
insertable
-into
```

现在读者已经了解了简单视图的局限性，下面将介绍如何使用视图连接多表。

14.3.2 更新复杂视图

单表视图确实很常用，不过读者遇到的许多视图也可能并非如此，定义里的基础查询的 from 子句中也可能包括多个表。例如，下面的视图连接 business 和 customer 两个表，进而可以轻易地查询到商业客户的所有数据：

```
CREATE VIEW business_customer_vw
 (cust_id,
  fed_id,
  address,
  city,
  state,
  postal_code,
  business_name,
  state_id,
  incorp_date
 )
AS
SELECT cst.cust_id,
  cst.fed_id,
  cst.address,
  cst.city,
  cst.state,
  cst.postal_code,
  bsn.name,
  bsn.state_id,
  bsn.incorp_date
FROM customer cst INNER JOIN business bsn
```

```
  ON cst.cust_id = bsn.cust_id
WHERE cust_type_cd = 'B';
```

读者可以使用这个视图更新 customer 表或者 business 表，具体如下：

```
mysql> UPDATE business_customer_vw
    -> SET postal_code = '99999'
    -> WHERE cust_id = 10;
Query OK, 1 row affected (0.09 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE business_customer_vw
    -> SET incorp_date = '2008-11-17'
    -> WHERE cust_id = 10;
Query OK, 1 row affected (0.11 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

第一个语句修改 customer.postal_code 列，第二个语句修改 business.incorp_date 列。读者可能会想：如果试图在单个语句中更新两个表的列，那么将会如何？下面我们来解决这个问题:

```
mysql> UPDATE business_customer_vw
    -> SET postal_code = '88888', incorp_date = '2008-10-31'
    -> WHERE cust_id = 10;
ERROR 1393 (HY000): Can not modify more than one base table through a join
view
'bank.business_customer_vw'
```

正如读者所见，只要读者不试图只使用一个语句，修改两个基础表还是可行的。现在让我们尝试向两个表中插入一个新客户（cust_id = 99）：

```
mysql> INSERT INTO business_customer_vw
   ->  (cust_id, fed_id, address, city, state, postal_code)
   -> VALUES (99, '04-9999999', '99 Main St.', 'Peabody', 'MA', '01975');
Query OK, 1 row affected (0.07 sec)

mysql> INSERT INTO business_customer_vw
   ->  (cust_id, business_name, state_id, incorp_date)
   -> VALUES (99, 'Ninety-Nine Restaurant', '99-999-999', '1999-01-01');
ERROR 1393 (HY000): Can not modify more than one base table through a join
view
'bank.business_customer_vw'
```

第一个语句试图向 customer 表中插入数据时成功了，但是第二个试图向 business 表中插入数据时抛出了一个错误，这是因为虽然两个表都包含 cust_id 列，但是视图定义中的 cust_id 列是映射到 customer.cust_id 列的，因此无法使用前面的视图定义向 business 表插入数据。

提示

Oracle 数据库和 SQL Server 也允许通过视图插入或修改数据。不过，其他数据库（如 MySQL）还会有些限制。如果打算写一些 PL/SQL 或 Transact-SQL 语句，那么读者可以使用一种称为替代触发器的功能，它允许读者从底层拦截作用于视图的 insert、update 和 delete 语句，再写包含这些变化的自定义代码。没有了这种类型的功能，非频繁应用中通过视图更新数据将会有太多的限制。

14.4 小测验

下列练习测试读者对视图的理解。完成习题后，参照附录 C 检查答案。

练习 14-1

创建一个视图，查询 employee 表并生成下列结果，要求不使用 where 子句。

```
+-----------------+------------------+
| supervisor_name | employee_name    |
+-----------------+------------------+
| NULL            | Michael Smith    |
| Michael Smith   | Susan Barker     |
| Michael Smith   | Robert Tyler     |
| Robert Tyler    | Susan Hawthorne  |
| Susan Hawthorne | John Gooding     |
| Susan Hawthorne | Helen Fleming    |
| Helen Fleming   | Chris Tucker     |
| Helen Fleming   | Sarah Parker     |
| Helen Fleming   | Jane Grossman    |
| Susan Hawthorne | Paula Roberts    |
| Paula Roberts   | Thomas Ziegler   |
| Paula Roberts   | Samantha Jameson |
| Susan Hawthorne | John Blake       |
| John Blake      | Cindy Mason      |
| John Blake      | Frank Portman    |
| Susan Hawthorne | Theresa Markham  |
| Theresa Markham | Beth Fowler      |
| Theresa Markham | Rick Tulman      |
+-----------------+------------------+
18 rows in set (1.47 sec)
```

练习 14-2

除了查询各分行开立的所有账户的余额，银行总裁还想要一张显示各分行名字及城市的报表。创建一个生成这些数据的视图。

第 15 章

元数据

除了存储用户插入数据库的所有数据，数据库服务器也需要创建数据库对象（表、视图、索引等）来保存这些数据。不必奇怪，数据库服务器在数据库里确实存储了这个信息。本章将讨论所谓的元数据是如何被存储的，它存储在哪里，如何访问它以及如何利用它构建灵活的系统。

15.1　关于数据的数据

元数据本质上是关于数据的数据。每次创建数据库对象时，数据库服务器需要记录各种各样的信息。例如，打算创建一个包含多列的表，它有主键约束、3 个索引以及 1 个外键约束，那么数据库服务器将需要保存下列信息：

- 表名；
- 表存储信息（表空间、初始大小等）；
- 存储引擎；
- 列名；
- 列数据类型；
- 默认列值；
- 非空列约束；
- 主键列；
- 主键名；
- 主键索引名；
- 索引名；

- 索引类型（B 树、位图）；
- 索引列；
- 索引列排序顺序（升序或降序）；
- 索引存储信息；
- 外键名；
- 外键列；
- 外键的关联表/列。

这个数据统称为数据字典或者系统目录。数据库服务器需要不断地保存这个数据，同时，为了验证和执行 SQL 语句它需要能够快速地检索数据。此外，数据库服务器必须保护这个数据只能被通过恰当的机制修改，比如 alter table 语句。

虽然已经存在用于服务器间进行元数据交换的标准，但是每个数据库服务器还是使用不同的机制提供元数据，例如：

- 一组视图，比如 Oracle 数据库的 user_tables 和 all_constraints 视图；
- 一组系统存储过程，比如 SQL Server 的 sp_tables 过程或者 Oracle 数据库的 dbms_metadata 包；
- 一种特殊数据库，比如 MySQL 的 information_schema 数据库。

除了这种带有 Sybase 系列数据库痕迹的系统存储过程， SQL Server 还包括一种称为 information_schema 的特殊模式，它会在每个数据库中自动提供。为了遵守 ANSI SQL:2003 标准，MySQL 和 SQL Server 都提供这个接口。本章剩下的部分将讨论 information_schema 对象。

15.2　信息模式

information_schema 数据库（或在 SQL Server 中为模式）里的所有可用对象是视图。不像在本书前面几章那样，我使用描述用途的方式说明各种表及视图的结构，information_schema 数据库中的视图是可以被检索的，因此它可以被编程使用（本章稍后有更多探讨）。下面的例子说明如何检索 bank 数据库里所有表的名字。

```
mysql> SELECT table_name, table_type
    -> FROM information_schema.tables
    -> WHERE table_schema = 'bank'
    -> ORDER BY 1;
+----------------------+------------+
| table_name           | table_type |
+--------------------  +----------  +
```

```
| account              | BASE TABLE |
| branch               | BASE TABLE |
| branch_activity_vw   | VIEW       |
| business             | BASE TABLE |
| business_customer_vw | VIEW       |
| customer             | BASE TABLE |
| customer_vw          | VIEW       |
| department           | BASE TABLE |
| employee             | BASE TABLE |
| employee_vw          | VIEW       |
| individual           | BASE TABLE |
| nh_customer_vw       | VIEW       |
| officer              | BASE TABLE |
| product              | BASE TABLE |
| product_type         | BASE TABLE |
| transaction          | BASE TABLE |
+----------------------+------------+
16 rows in set (0.02 sec)
```

除了在第 2 章中创建的各种表，结果中还显示了第 14 章中创建的几个视图。如果希望在结果中排除视图，可以简单地添加另一个条件到 where 子句：

```
mysql> SELECT table_name, table_type
    -> FROM information_schema.tables
    -> WHERE table_schema = 'bank' AND table_type = 'BASE TABLE'
    -> ORDER BY 1;
+--------------+------------+
| table_name   | table_type |
+--------------+------------+
| account      | BASE TABLE |
| branch       | BASE TABLE |
| business     | BASE TABLE |
| customer     | BASE TABLE |
| department   | BASE TABLE |
| employee     | BASE TABLE |
| individual   | BASE TABLE |
| officer      | BASE TABLE |
| product      | BASE TABLE |
| product_type | BASE TABLE |
| transaction  | BASE TABLE |
+--------------+------------+
11 rows in set (0.01 sec)
```

如果只对视图信息感兴趣，那么读者可以查询 information_schema.views。除了视图名称，读者还可以检索其他信息，比如说明视图是否可更新的标志：

```
mysql> SELECT table_name, is_updatable
    -> FROM information_schema.views
    -> WHERE table_schema = 'bank'
    -> ORDER BY 1;
```

```
+----------------------+-------------+
| table_name           | is_updatable |
+----------------------+-------------+
| branch_activity_vw   | NO          |
| business_customer_vw | YES         |
| customer_vw          | YES         |
| employee_vw          | YES         |
| nh_customer_vw       | YES         |
+----------------------+-------------+
5 rows in set (1.83 sec)
```

另外,只要查询足够小(MySQL 是 4 000 个字符或更少),读者就可以使用 view_definition 列检索视图的基础查询。

两个表的列信息和视图都是可用的。下面的查询显示了 account 表的列信息：

```
mysql> SELECT column_name, data_type, character_maximum_length char_max_len,
    ->   numeric_precision num_prcsn, numeric_scale num_scale
    -> FROM information_schema.columns
    -> WHERE table_schema = 'bank' AND table_name = 'account'
    -> ORDER BY ordinal_position;
+--------------------+----------+-------------+----------+--------+
| column_name        | data_type | char_max_len | num_prcsn |num_scale|
+--------------------+----------+-------------+----------+--------+
| account_id         | int      |        NULL |     10   |      0 |
| product_cd         | varchar  |          10 |   NULL   |   NULL |
| cust_id            | int      |        NULL |     10   |      0 |
| open_date          | date     |        NULL |   NULL   |   NULL |
| close_date         | date     |        NULL |   NULL   |   NULL |
| last_activity_date | date     |        NULL |   NULL   |   NULL |
| status             | enum     |           6 |   NULL   |   NULL |
| open_branch_id     | smallint |        NULL |      5   |      0 |
| open_emp_id        | smallint |        NULL |      5   |      0 |
| avail_balance      | float    |        NULL |     10   |      2 |
| pending_balance    | float    |        NULL |     10   |      2 |
+--------------------+----------+-------------+----------+--------+
11 rows in set (0.02 sec)
```

语句中包括 ordinal_position 列，只是说明按照列添加的顺序检索列。

读者可以通过检索 information_schema.statistics 视图获得某个表的索引信息。下面的查询检索 account 表中构建的索引信息：

```
mysql> SELECT index_name, non_unique, seq_in_index, column_name
    -> FROM information_schema.statistics
    -> WHERE table_schema = 'bank' AND table_name = 'account'
    -> ORDER BY 1, 3;
+---------------+------------+--------------+---------------+
| index_name    | non_unique | seq_in_index | column_name   |
+---------------+------------+--------------+---------------+
| acc_bal_idx   |          1 |            1 | cust_id       |
```

```
| acc_bal_idx     |          1 |            2 | avail_balance  |
| fk_a_branch_id  |          1 |            1 | open_branch_id |
| fk_a_emp_id     |          1 |            1 | open_emp_id    |
| fk_product_cd   |          1 |            1 | product_cd     |
| PRIMARY         |          0 |            1 | account_id     |
+-----------------+------------+--------------+----------------+
6 rows in set (0.09 sec)
```

account 表中总共有 5 个索引，其中一个有两列（acc_bal_idx），还有一个是唯一索引（PRIMARY）。

读者可以通过检索 information_schema.table_constraints 视图获得已经创建的不同类型的约束（外键、主键、唯一性）信息。下面的查询检索 bank 模式中的所有约束：

```
mysql> SELECT constraint_name, table_name, constraint_type
    -> FROM information_schema.table_constraints
    -> WHERE table_schema = 'bank'
    -> ORDER BY 3,1;
+--------------------+--------------+-----------------+
| constraint_name    | table_name   | constraint_type |
+--------------------+--------------+-----------------+
| fk_a_branch_id     | account      | FOREIGN KEY     |
| fk_a_cust_id       | account      | FOREIGN KEY     |
| fk_a_emp_id        | account      | FOREIGN KEY     |
| fk_b_cust_id       | business     | FOREIGN KEY     |
| fk_dept_id         | employee     | FOREIGN KEY     |
| fk_exec_branch_id  | transaction  | FOREIGN KEY     |
| fk_e_branch_id     | employee     | FOREIGN KEY     |
| fk_e_emp_id        | employee     | FOREIGN KEY     |
| fk_i_cust_id       | individual   | FOREIGN KEY     |
| fk_o_cust_id       | officer      | FOREIGN KEY     |
| fk_product_cd      | account      | FOREIGN KEY     |
| fk_product_type_cd | product      | FOREIGN KEY     |
| fk_teller_emp_id   | transaction  | FOREIGN KEY     |
| fk_t_account_id    | transaction  | FOREIGN KEY     |
| PRIMARY            | branch       | PRIMARY KEY     |
| PRIMARY            | account      | PRIMARY KEY     |
| PRIMARY            | product      | PRIMARY KEY     |
| PRIMARY            | department   | PRIMARY KEY     |
| PRIMARY            | customer     | PRIMARY KEY     |
| PRIMARY            | transaction  | PRIMARY KEY     |
| PRIMARY            | officer      | PRIMARY KEY     |
| PRIMARY            | product_type | PRIMARY KEY     |
| PRIMARY            | employee     | PRIMARY KEY     |
| PRIMARY            | business     | PRIMARY KEY     |
| PRIMARY            | individual   | PRIMARY KEY     |
| dept_name_idx      | department   | UNIQUE          |
+--------------------+--------------+-----------------+
26 rows in set (2.28 sec)
```

表 15-1 显示了 information_schema 的所有视图集，可用于 MySQL 6.0 版本。

表 15-1　information_schema 视图

视 图 名 称	提供的相关信息
Schemata	数据库
Tables	表和视图
Columns	表和视图的列
Statistics	索引
User_Privileges	模式权限分配
Schema_Privileges	数据库权限分配
Table_Privileges	表权限分配
Column_Privileges	列权限分配
Character_Sets	可用字符集
Collations	各字符集对照信息
Collation_Character_Set_Applicability	可用于校对的字符集
Table_Constraints	唯一、外键和主键约束
Key_Column_Usage	与每个键列相关的约束
Routines	存储例程（过程和函数 ）
Views	视图
Triggers	表触发器
Plugins	服务器插件程序
Engines	可用存储引擎
Partitions	表分区
Events	预定时间
Process_List	正在运行的进程
Referential_Constraints	外键
Global_Status	服务器状态信息
Session_Status	会话状态信息
Global_Variables	服务器状态变量
Session_Variables	会话状态变量
Parameters	存储过程和函数参数
Profiling	用户配置信息

有些视图是 MySQL 特有的，比如 engines、events 和 plugins，不过多数视图在 SQL Server 中也是可用的。如果读者正在使用 Oracle，那么请查阅在线“Oracle 数据库参考手册”获取 user_、 all_和 dba_ 这 3 个视图的相关信息。

15.3 使用元数据

前文中已经提到，拥有通过 SQL 查询检索模式对象相关信息的能力为许多有趣的应用提供了可能。本节介绍几种可以在程序中使用元数据的方法。

15.3.1 模式生成脚本

虽然有些项目组包括了监督数据库设计与执行的数据库设计者，但许多项目仍采用设计委员会的方法，此种情况下允许多人创建数据库对象。经过几周或几个月的开发，读者可能需要生成一个脚本，它能够创建小组已经部署的各种表、索引、视图等。虽然各种工具和程序可以生成这种类型的脚本，但是读者也可以自己查询 information_schema 视图，然后生成脚本。

举一个例子，假设要生成一个用于创建 bank.customer 表的视图。下面的命令用于创建表，它是我从创建范例数据库的脚本中截取的：

```
create table customer
 (cust_id integer unsigned not null auto_increment,
  fed_id varchar(12) not null,
  cust_type_cd enum('I','B') not null,
  address varchar(30),
  city varchar(20),
  state varchar(20),
  postal_code varchar(10),
  constraint pk_customer primary key (cust_id)
 );
```

利用程序语言（如 Transact-SQL 或 Java）无疑是很容易生成脚本的，但是本书是关于 SQL 知识的，所以我将会写一个单一查询生成 create table 语句。第一步是查询 information_schema.columns 表以检索表中哪些列的信息：

```
mysql> SELECT 'CREATE TABLE customer (' create_table_statement
    -> UNION ALL
    -> SELECT cols.txt
    -> FROM
    ->  (SELECT concat('  ',column_name, ' ', column_type,
    ->    CASE
    ->      WHEN is_nullable = 'NO' THEN ' not null'
    ->      ELSE ''
    ->    END,
    ->    CASE
    ->      WHEN extra IS NOT NULL THEN concat(' ', extra)
```

```
    ->      ELSE ''
    ->     END,
    ->     ',') txt
    ->   FROM information_schema.columns
    ->   WHERE table_schema = 'bank' AND table_name = 'customer'
    ->   ORDER BY ordinal_position
    ->  ) cols
    -> UNION ALL
    -> SELECT ')';
+-----------------------------------------------------+
| create_table_statement                              |
+-----------------------------------------------------+
| CREATE TABLE customer (                             |
|   cust_id int(10) unsigned not null auto_increment, |
|   fed_id varchar(12) not null ,                     |
|   cust_type_cd enum('I','B') not null ,             |
|   address varchar(30) ,                             |
|   city varchar(20) ,                                |
|   state varchar(20) ,                               |
|   postal_code varchar(10) ,                         |
| )                                                   |
+-----------------------------------------------------+
9 rows in set (0.04 sec)
```

现在，问题基本解决了。不过为了检索主键约束信息，还需要添加对 table_constraints 和 key_column_usage 两个视图的查询。

```
mysql> SELECT 'CREATE TABLE customer (' create_table_statement
    -> UNION ALL
    -> SELECT cols.txt
    -> FROM
    ->  (SELECT concat('  ',column_name, ' ', column_type,
    ->     CASE
    ->       WHEN is_nullable = 'NO' THEN ' not null'
    ->       ELSE ''
    ->     END,
    ->     CASE
    ->       WHEN extra IS NOT NULL THEN concat(' ', extra)
    ->       ELSE ''
    ->     END,
    ->     ',') txt
    ->   FROM information_schema.columns
    ->   WHERE table_schema = 'bank' AND table_name = 'customer'
    ->   ORDER BY ordinal_position
    ->  ) cols
    -> UNION ALL
    -> SELECT concat('  constraint primary key (')
    -> FROM information_schema.table_constraints
    -> WHERE table_schema = 'bank' AND table_name = 'customer'
    ->   AND constraint_type = 'PRIMARY KEY'
```

```
    -> UNION ALL
    -> SELECT cols.txt
    -> FROM
    ->  (SELECT concat(CASE WHEN ordinal_position > 1 THEN '   ,'
    ->     ELSE '    ' END, column_name) txt
    ->   FROM information_schema.key_column_usage
    ->   WHERE table_schema = 'bank' AND table_name = 'customer'
    ->     AND constraint_name = 'PRIMARY'
    ->   ORDER BY ordinal_position
    ->  ) cols
    -> UNION ALL
    -> SELECT '  )'
    -> UNION ALL
    -> SELECT ')';
+-------------------------------------------------------+
| create_table_statement                                |
+-------------------------------------------------------+
| CREATE TABLE customer (                               |
|   cust_id int(10) unsigned not null auto_increment,  |
|   fed_id varchar(12) not null ,                       |
|   cust_type_cd enum('I','B') not null ,               |
|   address varchar(30) ,                               |
|   city varchar(20) ,                                  |
|   state varchar(20) ,                                 |
|   postal_code varchar(10) ,                           |
|   constraint primary key (                            |
|     cust_id                                           |
|   )                                                   |
| )                                                     |
+-------------------------------------------------------+
12 rows in set (0.02 sec)
```

为了查看是否可以正确地生成语句，可把查询结果粘贴到 MySQL 工具中（将表名改为 customer2，这样不会干扰到其他表）：

```
mysql> CREATE TABLE customer2 (
    ->   cust_id int(10) unsigned not null auto_increment,
    ->   fed_id varchar(12) not null ,
    ->   cust_type_cd enum('I','B') not null ,
    ->   address varchar(30) ,
    ->   city varchar(20) ,
    ->   state varchar(20) ,
    ->   postal_code varchar(10) ,
    ->   constraint primary key (
    ->     cust_id
    ->   )
    -> );
Query OK, 0 rows affected (0.14 sec)
```

现在数据库中有了一个 customer2 表，语句执行成功！为了能编写一个查询，以生成良

好的 create table 语句，还需要做更多的工作（比如处理索引和外键索引约束），这里将其作为一个习题留给读者。

15.3.2 部署验证

许多组织允许有数据库维护窗口，在那里可以管理现有的数据库对象（比如添加、删除分区），以及部署新模式和新对象。在部署脚本运行后，推荐读者运行验证脚本以确保新的模式对象具有合适的列、索引、主键等。下面的查询返回 bank 模式中每个表的列数、索引数以及主键约束数（0 或 1）：

```
mysql> SELECT tbl.table_name,
    -> (SELECT count(*) FROM information_schema.columns clm
    ->  WHERE clm.table_schema = tbl.table_schema
    ->    AND clm.table_name = tbl.table_name) num_columns,
    -> (SELECT count(*) FROM information_schema.statistics sta
    ->  WHERE sta.table_schema = tbl.table_schema
    ->    AND sta.table_name = tbl.table_name) num_indexes,
    -> (SELECT count(*) FROM information_schema.table_constraints tc
    ->  WHERE tc.table_schema = tbl.table_schema
    ->    AND tc.table_name = tbl.table_name
    ->    AND tc.constraint_type = 'PRIMARY KEY') num_primary_keys
    -> FROM information_schema.tables tbl
    -> WHERE tbl.table_schema = 'bank' AND tbl.table_type = 'BASE TABLE'
    -> ORDER BY 1;
+--------------+-------------+-------------+------------------+
| table_name   | num_columns | num_indexes | num_primary_keys |
+--------------+-------------+-------------+------------------+
| account      |          11 |           6 |                1 |
| branch       |           6 |           1 |                1 |
| business     |           4 |           1 |                1 |
| customer     |           7 |           1 |                1 |
| department   |           2 |           2 |                1 |
| employee     |           9 |           4 |                1 |
| individual   |           4 |           1 |                1 |
| officer      |           7 |           2 |                1 |
| product      |           5 |           2 |                1 |
| product_type |           2 |           1 |                1 |
| transaction  |           8 |           4 |                1 |
+--------------+-------------+-------------+------------------+
11 rows in set (13.83 sec)
```

读者可以在部署前、后两次执行这个语句，最后，在宣布部署成功之前验证两个结果集之间的区别。

15.3.3 生成动态 SQL

一些语言（比如 Oracle 的 PL/SQL 和 Microsoft 的 Transact-SQL）是 SQL 语言的超集，这意味着除了常用程序结构（如“if-then-else”和“while”），它们还在语法中包括 SQL

语句。另外一些语言（比如 Java）包括与关系数据库连接的能力，但是在语法中不包括 SQL 语句，这意味着所有的 SQL 语句都必须被包含在一个字符串中。

因此，大多数数据库服务器（包括 SQL Server、Oracle 数据库和 MySQL）允许 SQL 语句以字符串形式提交。提交字符串给数据库引擎而不是使用它的 SQL 接口通常被称为动态 SQL 执行。例如，Oracle 的 PL/SQL 语言包括 execute immediate 命令，可用于提交字符串执行，SQL Server 则包括一个叫做 sp_executesql 的可以动态执行语句的系统存储过程。

MySQL 为动态 SQL 执行提供了 prepare、execute 和 deallocate 语句。下面是一个简单的例子：

```
mysql> SET @qry = 'SELECT cust_id, cust_type_cd, fed_id FROM customer';
Query OK, 0 rows affected (0.07 sec)

mysql> PREPARE dynsql1 FROM @qry;
Query OK, 0 rows affected (0.04 sec)
Statement prepared

mysql> EXECUTE dynsql1;
+---------+--------------+-------------+
| cust_id | cust_type_cd | fed_id      |
+---------+--------------+-------------+
|       1 | I            | 111-11-1111 |
|       2 | I            | 222-22-2222 |
|       3 | I            | 333-33-3333 |
|       4 | I            | 444-44-4444 |
|       5 | I            | 555-55-5555 |
|       6 | I            | 666-66-6666 |
|       7 | I            | 777-77-7777 |
|       8 | I            | 888-88-8888 |
|       9 | I            | 999-99-9999 |
|      10 | B            | 04-1111111  |
|      11 | B            | 04-2222222  |
|      12 | B            | 04-3333333  |
|      13 | B            | 04-4444444  |
|      99 | I            | 04-9999999  |
+---------+--------------+-------------+
14 rows in set (0.27 sec)
mysql> DEALLOCATE PREPARE dynsql1;
Query OK, 0 rows affected (0.00 sec)
```

set 语句只是简单地将字符串赋予变量 qry，之后 qry 将会被 prepare 语句提交给数据库引擎(为了解析、安全检查和优化)。在调用 execute 执行完语句后，必须再调用 deallocate prepare 关闭语句，释放执行中使用的所有数据库资源（如游标）。

通过在查询中包含占位符，也可以在运行时动态指定条件，下面的例子说明了如何执行这种查询：

```
mysql> SET @qry = 'SELECT product_cd, name, product_type_cd, date_offered,
date_retired FROM product WHERE product_cd = ?';
Query OK, 0 rows affected (0.00 sec)

mysql> PREPARE dynsql2 FROM @qry;
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql> SET @prodcd = 'CHK';
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE dynsql2 USING @prodcd;
+----------+----------------+---------------+-----------+-----------+
|product_cd | name             | product_type_cd | date_offered | date_retired |
+----------+----------------+---------------+-----------+-----------+
| CHK       | checking account | ACCOUNT         | 2004-01-01   | NULL         |
+----------+----------------+---------------+-----------+-----------+
1 row in set (0.01 sec)

mysql> SET @prodcd = 'SAV';
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE dynsql2 USING @prodcd;
+----------+----------------+---------------+-----------+-----------+
| product_cd| name             |product_type_cd |date_offered |date_retired|
+----------+----------------+---------------+-----------+-----------+
| SAV       | savings account | ACCOUNT         | 2004-01-01   | NULL         |
+----------+----------------+---------------+-----------+-----------+
1 row in set (0.00 sec)

mysql> DEALLOCATE PREPARE dynsql2;
Query OK, 0 rows affected (0.00 sec)
```

这次的查询包含了一个占位符（就是语句结尾的符号“？”），所以产品代码可以在运行时被提交。语句一旦被准备，就被提交两次，一次为产品代码“CHK”，另一次为产品代码“SAV”，之后语句被关闭。

读者可能想知道：这与元数据有什么关系呢？如果打算使用动态 SQL 查询表，那么为什么不使用元数据构建查询而是硬编码实现表的定义？下面的范例和前面的查询一样生成同样的动态 SQL 字符，但是它从视图 information_schema.columns 检索列名：

```
mysql> SELECT concat('SELECT ',
    ->   concat_ws(',', cols.col1, cols.col2, cols.col3, cols.col4,
    ->     cols.col5, cols.col6, cols.col7, cols.col8, cols.col9),
    ->   ' FROM product WHERE product_cd = ?')
    -> INTO @qry
    -> FROM
    ->  (SELECT
    ->     max(CASE WHEN ordinal_position = 1 THEN column_name
    ->       ELSE NULL END) col1,
```

```
    ->     max(CASE WHEN ordinal_position = 2 THEN column_name
    ->       ELSE NULL END) col2,
    ->     max(CASE WHEN ordinal_position = 3 THEN column_name
    ->       ELSE NULL END) col3,
    ->     max(CASE WHEN ordinal_position = 4 THEN column_name
    ->       ELSE NULL END) col4,
    ->     max(CASE WHEN ordinal_position = 5 THEN column_name
    ->       ELSE NULL END) col5,
    ->     max(CASE WHEN ordinal_position = 6 THEN column_name
    ->       ELSE NULL END) col6,
    ->     max(CASE WHEN ordinal_position = 7 THEN column_name
    ->       ELSE NULL END) col7,
    ->     max(CASE WHEN ordinal_position = 8 THEN column_name
    ->       ELSE NULL END) col8,
    ->     max(CASE WHEN ordinal_position = 9 THEN column_name
    ->       ELSE NULL END) col9
    ->   FROM information_schema.columns
    ->   WHERE table_schema = 'bank' AND table_name = 'product'
    ->   GROUP BY table_name
    ->  ) cols;
Query OK, 1 row affected (0.02 sec)

mysql> SELECT @qry;
+---------------------------------------------------------------------
---------------------+
| @qry
                     |
+---------------------------------------------------------------------
---------------------+
| SELECT product_cd,name,product_type_cd,date_offered,date_retired
FROM product
WHERE product_cd = ? |
+---------------------------------------------------------------------
---------------------+
1 row in set (0.00 sec)

mysql> PREPARE dynsql3 FROM @qry;
Query OK, 0 rows affected (0.01 sec)
Statement prepared

mysql> SET @prodcd = 'MM';
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE dynsql3 USING @prodcd;
+----------+----------------------+-----------------+--------------+--------------+
| product_cd | name                 | product_type_cd | date_offered | date_retired |
+----------+----------------------+-----------------+--------------+--------------+
| MM       | money market account | ACCOUNT         | 2004-01-01   | NULL         |
+----------+----------------------+-----------------+--------------+--------------+
1 row in set (0.00 sec)
```

```
mysql> DEALLOCATE PREPARE dynsql3;
Query OK, 0 rows affected (0.00 sec)
```

这个查询以 product 表的前 9 列为核心，先使用 concat 和 concat_ws 函数构建查询字符串，然后将字符串赋予变量 qry，最后查询字符串像前面的查询一样被执行。

提示

通常，使用包括循环结构的过程语言（如 Java、PL/SQL、Transact-SQL 或者 MySQL 的存储过程语言）生成查询会更好一点。不过，我想说明纯 SQL 范例，但此种情况下的循环实现较为繁杂，因此不得不限制检索的列数至一个合理的数字，在本例中这个数字是 9。

15.4　小测验

下面的练习测试读者对元数据的理解。完成习题后，请参考附录 C 检查答案。

练习 15-1

编写一个查询，列出 bank 列中的所有索引，要求结果包括表名。

练习 15-2

编写一个查询，生成的结果可以用于创建 bank.employee 表的所有索引。要求结果形式如下：

```
"ALTER TABLE <table_name> ADD INDEX <index_name> (<column_list>)"
```

附录 A

示例数据库的 ER 图

图 A-1 是本书所用示例数据库的实体联系图（ER 图）。顾名思义，它是用于描述数据库中具有外键联系的表和实体的。下面几点可以帮助读者理解这个概念。

- 每个矩形代表一个表，左上角是表的名字，主键列与非键列被一条横线隔开列出，其中非键列位于横线之下，外键列则以“(FK)”标记出来。
- 表之间的直线代表外键关系。直线两端的数字代表允许的数量，它可能是 0、1 或者多个。例如，考察账号表与产品表之间的关系，读者就会了解一个账号只能属于一个产品，但是一个产品可能有 0、1 或者多个账号。

参阅 http://en.wikipedia.org/wiki/Entity-relationship_model 了解实体联系模型更多的相关信息。

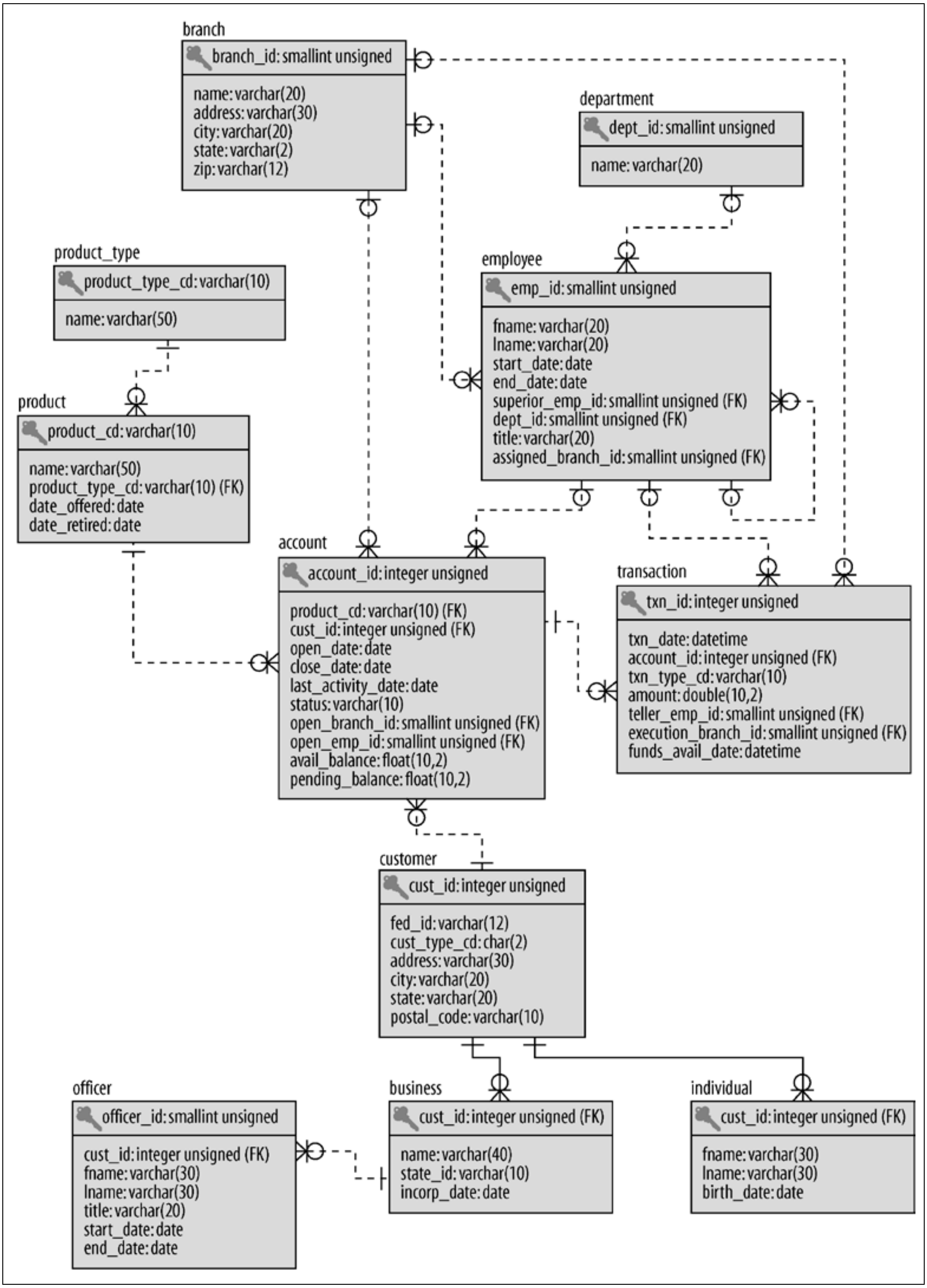

branch
branch_id: smallint unsigned
name: varchar(20)
address: varchar(30)
city: varchar(20)
state: varchar(2)
zip: varchar(12)
department
dept_id: smallint unsigned
name: varchar(20)
product_type
product_type_cd: varchar(10)
name: varchar(50)
employee
emp_id: smallint unsigned
fname: varchar(20)
lname: varchar(20)
start_date: date
end_date: date
superior_emp_id: smallint unsigned (FK)
dept_id: smallint unsigned (FK)
title: varchar(20)
assigned_branch_id: smallint unsigned (FK)
product
product_cd: varchar(10)
name: varchar(50)
product_type_cd: varchar(10) (FK)
date_offered: date
date_retired: date
account
account_id: integer unsigned
product_cd: varchar(10) (FK)
cust_id: integer unsigned (FK)
open_date: date
close_date: date
last_activity_date: date
status: varchar(10)
open_branch_id: smallint unsigned (FK)
open_emp_id: smallint unsigned (FK)
avail_balance: float(10,2)
pending_balance: float(10,2)
transaction
txn_id: integer unsigned
txn_date: datetime
account_id: integer unsigned (FK)
txn_type_cd: varchar(10)
amount: double(10,2)
teller_emp_id: smallint unsigned (FK)
execution_branch_id: smallint unsigned (FK)
funds_avail_date: datetime
customer
cust_id: integer unsigned
fed_id: varchar(12)
cust_type_cd: char(2)
address: varchar(30)
city: varchar(20)
state: varchar(20)
postal_code: varchar(10)
officer
officer_id: smallint unsigned
cust_id: integer unsigned (FK)
fname: varchar(30)
lname: varchar(30)
title: varchar(20)
start_date: date
end_date: date
business
cust_id: integer unsigned (FK)
name: varchar(40)
state_id: varchar(10)
incorp_date: date
individual
cust_id: integer unsigned (FK)
fname: varchar(30)
lname: varchar(30)
birth_date: date

图 A-1　ER 图

附录 B

MySQL 对 SQL 语言的扩展

本书的所有示例都使用了 MySQL 服务器，因此我认为在附录介绍 MySQL 的 SQL 语言扩展对计划进一步学习 MySQL 的读者是有益的。本附录主要探讨 MySQL 中在某些情况下特别有用的扩展，比如 select、insert、update 和 delete 语句。

B.1 扩展 select 语句

MySQL 对 select 语句的完善包括另外两个子句，下面逐一进行讨论。

B.1.1 limit 子句

某些情况下，读者可能并非对查询返回的所有结果都感兴趣。例如，构造一个查询返回银行所有柜员及其开立的账户数，但是如果读者只是为了查询前三位柜员，从而对其进行奖励，那么就不必知道谁是第四名、第五名，等等。为了解决这种问题，MySQL 的 select 语句包括了 limit 子句，它允许读者限制查询返回的行数。

为了展示 limit 子句的用法，首先构造一个查询，返回银行中每个柜员开立的账户数：

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          13 |        3 |
|          16 |        6 |
+-------------+----------+
4 rows in set (0.31 sec)
```

结果显示 4 个不同的柜员开立的账户数。如果想限制结果集到仅 3 条记录，那么读者可以添加一个 limit 子句指定只有 3 条记录被返回：

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> LIMIT 3;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          13 |        3 |
+-------------+----------+
3 rows in set (0.06 sec)
```

由于 limit 子句（上面查询的第四行）的限制，现在结果集刚好包括 3 条记录，而第四名柜员（员工 ID 为 16）则被从结果集中丢弃。

组合 limit 子句和 order by 子句

虽然前面的查询返回 3 个记录，不过有一个小问题，即读者不能描述究竟对这 4 个记录中的哪 3 个感兴趣。如果要查找 3 个特定的记录，比如开立账户数最多的 3 个，那么读者就需要结合 order by 子句使用 limit 子句，具体如下：

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> ORDER BY how_many DESC
    -> LIMIT 3;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          16 |        6 |
+-------------+----------+
3 rows in set (0.03 sec)
```

这个查询与前一个的不同之处在于现在将 limit 子句应用到有序集，从而导致开立账户数最多的 3 个柜员被包含在最终的结果集中。除非读者只是对记录中任意样本感兴趣，否则，通常都需要组合使用 order by 子句和 limit 子句。

提示

由于 limit 子句应用在所有过滤、分组和排序动作完成后，因此它永远不会改变选择语句的结果，而只是限制查询返回记录的数目。

Limit 子句中可选的第二个参数

假设读者的目标不是查找前 3 位柜员，而是前两位之外的其他所有柜员（例如，银行目的不是奖励表现最好的，而是要送一些表现不好的去参加自信训练）。对于这些情况，limit 语句允许可选的第二个参数，此时第一个指定结果集的起始记录，第二个参数指定结果集包含的记录数。指定起始记录时，请记住 MySQL 指定第一个记录序号为 0。因此，如果读者希望查找第三个表现好的人，那么可以如下实现：

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> ORDER BY how_many DESC
    -> LIMIT 2, 1;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|          16 |        6 |
+-------------+----------+
1 row in set (0.00 sec)
```

在上面的例子中，第 0 个和第 1 个记录被丢弃了，因而结果集是从第 2 个记录开始的。由于 limit 子句的第二个参数是 1，所以结果集只包含 1 个记录。

如果读者希望从第 2 个位置开始，并且包括所有剩下的记录，那么可以使 limit 子句中的第二个参数足够大到超过剩下的记录数。如果不知道多少柜员开立新账户，那么读者可以如下查找前两个以外的所有柜员：

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> ORDER BY how_many DESC
    -> LIMIT 2, 999999999;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|          16 |        6 |
|          13 |        3 |
+-------------+----------+
2 rows in set (0.00 sec)
```

在这个查询中，第 0 个和第 1 个被丢弃了，并且从第 2 个记录起的 999 999 999 个记录被包含在结果集中（虽然这种情况下只有两个记录，但是记录数指定的大一点比较好，否则可能会出现由于估计不足而导致遗漏合法记录的情况）。

排名查询

包含 order by 子句和 limit 子句的查询允许将数据进行排名，故这种查询可以称为排名

查询。虽然我刚才描述的是如何依据开立的账户数对柜员进行排名，但是排名查询同样可以应用于各种不同的商业问题，例如：

- 我们 2005 年的前 5 名销售员是谁?
- 棒球历史上第三大全垒打是谁?
- 下一本 1998 年最畅销的书是什么?
- 我们最滞销口味的两种冰激凌是什么?

至此，我们已经展示了如何查找前三位的柜员，第三个优秀柜员以及前两个柜员以外的所有其他柜员。如果打算做一些相似的例子（比如查找最差表现者），那么只需要反转分类顺序使结果从最低账户数到最高账户数产生，例如：

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> ORDER BY how_many ASC
    -> LIMIT 2;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|          13 |        3 |
|          16 |        6 |
+-------------+----------+
2 rows in set (0.24 sec)
```

通过简单的改变排列顺序(从 ORDER BY how_many DESC 变为 ORDER BY how_many ASC)，现在查询即可返回表现最差的两个柜员。因此，通过使用带有升序排列条件或降序排列条件的 limit 子句，读者可以构造排名查询来解决大多数类型的商业问题。

B.1.2 into outfile 子句

如果希望将查询语句的结果写到一个文件中，读者可以突出显示查询结果，然后将其复制到缓存，最后粘贴到自己喜欢的编辑器。然而，如果查询结果集非常大或者查询从一个脚本执行，那么就需要一个方法自动将结果写入文件。为了解决这种问题，MySQL 包含了 into outfile 子句，允许只提供一个文件名，即可将结果写到这个文件中。下面的例子将查询的结果写到 C:\temp 目录下的一个文件中：

```
mysql> SELECT emp_id, fname, lname, start_date
    -> INTO OUTFILE 'C:\\TEMP\\emp_list.txt'
    -> FROM employee; Query OK, 18 rows affected (0.20 sec)
```

提示

如果读者还记得第 7 章中的内容，那么就会知道反斜线用于字符串中实现字符转义。对于一个 Windows 系列的用户，构建路径时需要单行输入两个反斜线。

结果没有输出到屏幕上，而是写进了文件 emp_list.txt，内容如下：

```
1    Michael    Smith    2001-06-22
2    Susan    Barker    2002-09-12
3    Robert    Tyler    2000-02-09
4    Susan    Hawthorne    2002-04-24
...
16    Theresa    Markham    2001-03-15
17    Beth    Fowler    2002-06-29
18    Rick    Tulman    2002-12-12
```

默认格式在列间使用制表符（'\t'）隔开，而记录间使用换行符（'\n'）隔开。如果读者希望对数据格式有更多的控制,那么也可以用其他一些子句来协助 into outfile 子句实现。例如，要实现数据以竖线分隔格式保存到文件中，可以使用 fields 子句要求每列之间使用字符'|'隔开，具体如下：

```
mysql> SELECT emp_id, fname, lname, start_date
    -> INTO OUTFILE 'C:\\TEMP\\emp_list_delim.txt'
    ->  FIELDS TERMINATED BY '|'
    -> FROM employee; Query OK, 18 rows affected (0.02 sec)
```

提示

使用 into outfile 子句时，MySQL 不允许覆盖现有文件，因而读者如果多次运行同一个查询，就需要移除存在的文件。

文件 emp_list_delim.txt 的内容如下：

```
1|Michael|Smith|2001-06-22
2|Susan|Barker|2002-09-12
3|Robert|Tyler|2000-02-09
4|Susan|Hawthorne|2002-04-24
...
16|Theresa|Markham|2001-03-15
17|Beth|Fowler|2002-06-29
18|Rick|Tulman|2002-12-12
```

除了竖线分隔格式，读者可能还需要逗号分隔格式，此时要使用 fields terminated by ','子句来实现。然而，如果写入的数据包括字符串，那么使用逗号作为域分隔符被证实是有问题的，因为逗号比竖线看上去更像是字符串的一部分。思考下面的查询，它将一个数字和两个字符串写入文件 commal.txt，并且用逗号隔开：

```
mysql> SELECT data.num, data.str1, data.str2
    -> INTO OUTFILE 'C:\\TEMP\\comma1.txt'
    ->  FIELDS TERMINATED BY ','
    -> FROM
    ->  (SELECT 1 num, 'This string has no commas' str1,
```

```
    ->    'This string, however, has two commas' str2) data;
Query OK, 1 row affected (0.04 sec)
```

读者可能会想，由于输出文件的第三列（str2）是一个包含逗号的字符串，如果程序试图读取 commal.txt 文件，将行解析到列，那么将会遇到问题。不过，MySQL 已经为此做好准备了。下面就是 commal.txt 的内容：

```
1,This string has no commas,This string\, however\, has two commas
```

第三列中的字符串内嵌逗号已经被反斜线转义了。如果使用竖线分隔格式，逗号则不会被转义，这是因为不会有混淆。如果希望使用另一个不同的分隔符，比如使用另一种逗号，那么读者可以使用 fields terminated by 子句指定用于输出文件的分隔符。

除了指定列分隔符，读者还可以为数据文件不同的记录指定分隔符。如果不喜欢换行符分隔输出文件中的不同记录，那么读者可以使用 lines 子句指定另一种分隔符，具体如下：

```
mysql> SELECT emp_id, fname, lname, start_date
    -> INTO OUTFILE 'C:\\TEMP\\emp_list_atsign.txt'
    ->  FIELDS TERMINATED BY '|'
    ->  LINES TERMINATED BY '@'
    -> FROM employee;
Query OK, 18 rows affected (0.03 sec)
```

因为记录间没有使用换行符隔开，所以文件 emp_list_atsign.txt 的内容看上去像一个长的字符串，其中的记录被字符‘@’隔开：

```
1|Michael|Smith|2001-06-22@2|Susan|Barker|2002-09-12@3|Robert|Tyler|2000-02-
09@4|Susan|Hawthorne|2002-04-24@5|John|Gooding|2003-11-14@6|Helen|Fleming|2004-03-
17@7|Chris|Tucker|2004-09-15@8|Sarah|Parker|2002-12-02@9|Jane|Grossman|2002-05-
03@10|Paula|Roberts|2002-07-27@11|Thomas|Ziegler|2000-10-23@12|Samantha|Jameson|2003-
01-08@13|John|Blake|2000-05-11@14|Cindy|Mason|2002-08-09@15|Frank|Portman|2003-04-
01@16|Theresa|Markham|2001-03-15@17|Beth|Fowler|2002-06-29@18|Rick|Tulman|2002-12-12@
```

读者如果需要生成数据，用于电子表格程序，或者在自己的组织内外传送，那么 into outfile 应该能够提供足够的弹性，以实现需要的任何文件格式。

B.2 组合 insert/update 语句

假设要求读者创建一个表以获取哪些客户访问了哪些分行这个信息。这个表要包括客户 ID 列、分行 ID 列和日期列（描述客户访问该支行的最后时间）。无论客户何时访问分行，表中都会增加一列，但是如果客户已经存在，那么只需简单地对存在的行的 datetime 列进行修改。下面是表的定义:

```
CREATE TABLE branch_usage
 (branch_id SMALLINT UNSIGNED NOT NULL,
  cust_id INTEGER UNSIGNED NOT NULL,
  last_visited_on DATETIME,
  CONSTRAINT pk_branch_usage PRIMARY KEY (branch_id, cust_id)
 );
```

除了 3 个列定义，branch_uase 表还在 branch_id 和 cust_id 上定义了主键约束。因此，服务器将拒绝任何分行/客户对已经存在的行加入到表中。

假定表已确定如此，ID 为 5 的客户在第一周访问主分行（分行 ID 为 1）3 次。第一次访问后，由于不存在客户 ID 为 5、分行 ID 为 1 的记录，因而没有读者可以向 branch_usage 表中插入记录。

```
mysql> INSERT INTO branch_usage (branch_id, cust_id, last_visited_on)
    -> VALUES (1, 5, CURRENT_TIMESTAMP());
Query OK, 1 row affected (0.02 sec)
```

然而，下一次这个客户访问同一个分行时，读者需要更新一个存在的记录而不是插入一个新记录；否则，将会出现下面的错误：

```
ERROR 1062 (23000): Duplicate entry '1-5' for key 1
```

为了避免这种错误，读者可以查询 branch_usage 表以判断一个指定的客户/分行对是否存在，然后插入一个记录（如果没有记录被发现）或者更新一个存在的记录（如果已经存在）。不过，为了解决这个麻烦，MySQL 设计者扩展了 insert 语句，从而允许读者在 insert 语句由于重复键而失败时修改一列或多列。下面的语句命令服务器在给定的客户和分行已存在于 branch_usage 表中时修改 last_visited_on 列：

```
mysql> INSERT INTO branch_usage (branch_id, cust_id, last_visited_on)
    -> VALUES (1, 5, CURRENT_TIMESTAMP())
    -> ON DUPLICATE KEY UPDATE last_visited_on = CURRENT_TIMESTAMP();
Query OK, 2 rows affected (0.02 sec)
```

on duplicate key 子句允许这个语句在每次 ID 为 5 的客户到 ID 为 1 的分行开展业务时都执行。如果运行 100 次，那么第一个执行的结果是向表中添加一行，剩下的 99 次则是将 last_visited_on 列修改为当前时间。这种类型的操作通常被称为更新插入（upsert），因为它是 update 语句和 insert 语句的组合。

替换 repalce 命令

在 MySQL 4.1 之前的版本中，更新插入操作都是使用 replace 命令完成的。replace 是一个专有语句，它实现在插入前将已经存在相同主键的行删除。如果使用 4.1 及其之后的版本，就可以在实现更新插入时选择 replace 命令或者 insert…on duplicate key 命令。

不过，当遇到重复主键时，replace 命令执行删除操作，但是如果读者使用 InnoDB 存储引擎和启用外键约束，就会产生连锁反应。如果创建了带有 on delete cascade 选项

的约束，那么其他表中的行也可能随着 replace 命令删除目标表中的行而被自动删除。正是因为这个原因，在 insert 语句中使用 on duplicate key 通常被视为比 replace 命令更安全。

B.3 按排序更新和删除

本附录前面已经演示了如何在 order by 子句中使用 limit 子句编写产生排序的查询，比如获取账户开户数排在前 3 位的柜员。MySQL 还允许在 update 和 delete 语句中使用 limit 和 order by 子句，因此可以实现在表中根据次序修改或删除特定的行。举例来说，假设你需要从包含顾客在网上银行的登录信息的表中删除一些记录，该表记录了顾客 ID 和登录的日期/时间，如下所示。

```
CREATE TABLE login_history
 (cust_id INTEGER UNSIGNED NOT NULL,
  login_date DATETIME,
  CONSTRAINT pk_login_history PRIMARY KEY (cust_id, login_date)
 );
```

下面的语句为 login_history 表增加一些数据，这些数据来源于 account 表和 customer 表的交叉连接，并使用 account 的 open_date 列作为产生登录日期的基数：

```
mysql> INSERT INTO login_history (cust_id, login_date)
    -> SELECT c.cust_id,
    ->   ADDDATE(a.open_date, INTERVAL a.account_id * c.cust_id HOUR)
    -> FROM customer c CROSS JOIN account a;
Query OK, 312 rows affected (0.03 sec)
Records: 312  Duplicates: 0  Warnings: 0
```

现在该表含有 312 行相关的随机数据。下面的任务就是每月查看一次 login_history 表，并为经理创建一个显示在线银行系统用户的报表，然后保留该表最近 50 条记录并删除其他所有的记录。实现此目标的方法之一就是使用 order by 和 limit 编写查询，以找出前 50 个最近登录的记录，如下所示。

```
mysql> SELECT login_date
    -> FROM login_history
    -> ORDER BY login_date DESC
    -> LIMIT 49,1;
+---------------------+
| login_date          |
+---------------------+
| 2004-07-02 09:00:00 |
+---------------------+
1 row in set (0.00 sec)
```

借助此信息，可以构建 delete 语句，以删除所有 login_date 列小于上面查询所返回的日

期的行：

```
mysql> DELETE FROM login_history
    -> WHERE login_date < '2004-07-02 09:00:00';
Query OK, 262 rows affected (0.02 sec)
```

该表现在只包含了 50 个最近登录的记录。此外，使用 MySQL 的扩展语法，可以在单个 delete 语句中使用 limit 与 order by 子句中实现同样的结果。在原始的带有 312 行的 login_history 表中，运行下面的语句：

```
mysql> DELETE FROM login_history
    -> ORDER BY login_date ASC
    -> LIMIT 262;
Query OK, 262 rows affected (0.05 sec)
```

使用该语句，表中的行将根据 login_date 列以升序排列，然后删除前 262 行，保留 50 个最近的行。

提示

本例不得不事先知道表中原有的行数以构建 limit 子句(312 原始行-50 保留行=262 需删除行)。下面的做法可能更好一些，即将行以降序排序并告知服务器跳过前 50 行，再删除其余的行:

```
DELETE FROM login_history
ORDER BY login_date DESC
LIMIT 49, 9999999;
```

不过 MySQL 并不允许在 delete 或 update 语句中使用 limit 子句时提供第二个参数。

除了删除数据，还可以在修改数据时使用 limit 和 order by 子句。例如，银行决定为 10 个开户时间最久的账户增加$100 以奖励他们的忠诚度，可以采用下面的方法：

```
mysql> UPDATE account
    -> SET avail_balance = avail_balance + 100
    -> WHERE product_cd IN ('CHK', 'SAV', 'MM')
    -> ORDER BY open_date ASC
    -> LIMIT 10;
Query OK, 10 rows affected (0.06 sec)
Rows matched: 10  Changed: 10  Warnings: 0
```

该语句根据开户日期以升序方式对账户排序，并修改前 10 条记录，即储蓄时间最长的前 10 个账户。

B.4 多表更新与删除

在某些条件下，或许需要从几个表中修改或删除数据以完成给定的任务。例如，当你发现银行的数据库中包含了在系统测试时添加的虚假客户时，就需要从 account、customer 和 individual 表中删除数据。

提示

这里将分别为 account 表、customer 表和 individual 表创建克隆表，其表名分别为 account2、customer2 和 individual2。这样做的目的是保护示例数据不被更改以及避免与其他表（后面将介绍）之间产生外键约束被破坏的问题。下面是用于创建 3 个克隆表的 create table 语句：

```
CREATE TABLE individual2 AS
SELECT * FROM individual;
CREATE TABLE customer2 AS
SELECT * FROM customer;
CREATE TABLE account2 AS
SELECT * FROM account;
```

如果虚假顾客的 ID 为 1，那么可以为 3 个表分别创建 3 条独立的 delete 语句：

```
DELETE FROM account2
WHERE cust_id = 1;
DELETE FROM customer2
WHERE cust_id = 1;
DELETE FROM individual2
WHERE cust_id = 1;
```

不过，除了编写独立的 delete 语句，MySQL 还允许编写单条多重删除语句完成同样的工作，本例中的语句如下：

```
mysql> DELETE account2, customer2, individual2
    -> FROM account2 INNER JOIN customer2
    ->   ON account2.cust_id = customer2.cust_id
    ->   INNER JOIN individual2
    ->   ON customer2.cust_id = individual2.cust_id
    -> WHERE individual2.cust_id = 1;
Query OK, 5 rows affected (0.02 sec)
```

该语句一共删除了 5 行，其中 individual2 和 customer2 表各有 1 行，account2 表有 3 行（ID 为 1 的客户具有 3 个账户）。该语句包含 3 个独立的子句：

delete

指定要删除数据的表。

from

指定用于定位被删除数据的表。该子句的格式、功能与 select 语句中的 from 子句完全一样，并且这里出现的表不一定被包含在 delete 子句中。

where

包含用于定位被删除行的过滤条件。

多表删除语句看起来与 select 语句很相似，只不过使用 delete 子句替代了 select 子句。

如果使用多表删除语句的格式从单个表中删除行，那么它们之间的差别几乎看不出来。例如，下面的 select 语句查找所有 John Hayward 拥有的账户 ID：

```
mysql> SELECT account2.account_id
    -> FROM account2 INNER JOIN customer2
    ->   ON account2.cust_id = customer2.cust_id
    ->   INNER JOIN individual2
    ->   ON individual2.cust_id = customer2.cust_id
    -> WHERE individual2.fname = 'John'
    ->   AND individual2.lname = 'Hayward';
+------------+
| account_id |
+------------+
|          8 |
|          9 |
|         10 |
+------------+
3 rows in set (0.01 sec)
```

在查看结果之后，如果需要从 account2 表中删除 John 的 3 个账户，就在前一个查询语句中用针对 account2 表的 delete 子句替换 select 语句：

```
mysql> DELETE account2
    -> FROM account2 INNER JOIN customer2
    ->   ON account2.cust_id = customer2.cust_id
    ->   INNER JOIN individual2
    ->   ON customer2.cust_id = individual2.cust_id
    -> WHERE individual2.fname = 'John'
    ->   AND individual2.lname = 'Hayward';
Query OK, 3 rows affected (0.01 sec)
```

事实上，在多表 delete 语句中，对于 delete 和 from 子句的用法，还有更好的做法。上述语句的功能与下面的单表 delete 语句完全一样，它使用子查询来确定 John Hayward 的 customer ID：

```
DELETE FROM account2
WHERE cust_id =
 (SELECT cust_id
  FROM individual2
  WHERE fname = 'John' AND lname = 'Hayward';
```

当使用多表 delete 语句从单个表中删除数据时，可以自由选择与查询语法类似的包含连接的格式，或者是带有子查询的传统的 delete 语句。多表 delete 语句的真正能力体现在它能够在单个语句中删除多个表中的数据，如本节第一例查询语句所示。

除了可以从多个表中删除行，MySQL 还允许使用多表 update 语句同时修改多个表中的行。假如我们的银行需要和另一家银行合并，而这两家银行的数据库中含有重叠的 customer ID。因此经理决定将数据库中的每个 customer ID 增加 10 000，以便第二家银行的数据能够安全地被导入。下面的语句显示了如何在单个语句中修改 individual2 表、

customer2 表和 account2 表中 customer ID 为 3 的记录 ID：

```
mysql> UPDATE individual2 INNER JOIN customer2
    ->   ON individual2.cust_id = customer2.cust_id
    ->   INNER JOIN account2
    ->   ON customer2.cust_id = account2.cust_id
    -> SET individual2.cust_id = individual2.cust_id + 10000,
    ->   customer2.cust_id = customer2.cust_id + 10000,
    ->   account2.cust_id = account2.cust_id + 10000
    -> WHERE individual2.cust_id = 3;
Query OK, 4 rows affected (0.01 sec)
Rows matched: 5  Changed: 4  Warnings: 0
```

该语句一共修改了 4 行，individual2 表和 customer 表各 1 行，以及 account2 表中的两行。多表 update 语法与单表 update 非常相似，除了前者的 update 子句中包含了多个表以及相应的连接条件而不是只包含单个表名。与单表 update 语句一样，多表 update 语句同样包含了 set 子句，不同之处在于任何 update 子句中所引用的表都可以通过 set 子句进行修改。

提示

如果使用的是 InnoDB 存储引擎，并且表之间带有外键约束，那么将不能使用多表 delete 和 update 语句，这是因为该引擎并不保证数据修改能够按照不破坏约束的次序执行。这时应该使用多个执行次序正确的单表语句，以保证外键约束不被违反。

附录 C

练习答案

第 3 章

3-1

获取所有银行雇员的 employee ID、名字（first name）和姓氏（last name），并先后根据姓氏和名字进行排序。

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> ORDER BY lname, fname;
+--------+----------+-----------+
| emp_id | fname    | lname     |
+--------+----------+-----------+
|      2 | Susan    | Barker    |
|     13 | John     | Blake     |
|      6 | Helen    | Fleming   |
|     17 | Beth     | Fowler    |
|      5 | John     | Gooding   |
|      9 | Jane     | Grossman  |
|      4 | Susan    | Hawthorne |
|     12 | Samantha | Jameson   |
|     16 | Theresa  | Markham   |
|     14 | Cindy    | Mason     |
|      8 | Sarah    | Parker    |
|     15 | Frank    | Portman   |
|     10 | Paula    | Roberts   |
|      1 | Michael  | Smith     |
|      7 | Chris    | Tucker    |
|     18 | Rick     | Tulman    |
|      3 | Robert   | Tyler     |
|     11 | Thomas   | Ziegler   |
+--------+----------+-----------+
18 rows in set (0.01 sec)
```

3-2

获取所有状态为 'ACTIVE' 以及可用余额大于$2 500 的账户的 account ID、customer ID 和可用余额（available balance）。

```
mysql> SELECT account_id, cust_id, avail_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    ->   AND avail_balance > 2500;
+------------+---------+---------------+
| account_id | cust_id | avail_balance |
+------------+---------+---------------+
|          3 |       1 |       3000.00 |
|         10 |       4 |       5487.09 |
|         13 |       6 |      10000.00 |
|         14 |       7 |       5000.00 |
|         15 |       8 |       3487.19 |
|         18 |       9 |       9345.55 |
|         20 |      10 |      23575.12 |
|         22 |      11 |       9345.55 |
|         23 |      12 |      38552.05 |
|         24 |      13 |      50000.00 |
+------------+---------+---------------+
10 rows in set (0.00 sec)
```

3-3

针对 account 表编写查询，以返回开设过账户的雇员 ID（使用 account.open_emp_id 列），并且结果集中每个独立的雇员只包含一行数据。

```
mysql> SELECT DISTINCT open_emp_id
    -> FROM account;
+-------------+
| open_emp_id |
+-------------+
|           1 |
|          10 |
|          13 |
|          16 |
+-------------+
4 rows in set (0.00 sec)
```

3-4

为下面的多数据集查询语句填空（用<#>标记的地方），以获取所显示的结果。

```
mysql> SELECT p.product_cd, a.cust_id, a.avail_balance
    -> FROM product p INNER JOIN account <1>
    ->   ON p.product_cd = <2>
```

```
    -> WHERE p.<3> = 'ACCOUNT'
    -> ORDER BY <4>, <5>;
+------------+---------+--------------+
| product_cd | cust_id | avail_balance |
+------------+---------+--------------+
| CD         |       1 |      3000.00 |
| CD         |       6 |     10000.00 |
| CD         |       7 |      5000.00 |
| CD         |       9 |      1500.00 |
| CHK        |       1 |      1057.75 |
| CHK        |       2 |      2258.02 |
| CHK        |       3 |      1057.75 |
| CHK        |       4 |       534.12 |
| CHK        |       5 |      2237.97 |
| CHK        |       6 |       122.37 |
| CHK        |       8 |      3487.19 |
| CHK        |       9 |       125.67 |
| CHK        |      10 |     23575.12 |
| CHK        |      12 |     38552.05 |
| MM         |       3 |      2212.50 |
| MM         |       4 |      5487.09 |
| MM         |       9 |      9345.55 |
| SAV        |       1 |       500.00 |
| SAV        |       2 |       200.00 |
| SAV        |       4 |       767.77 |
| SAV        |       8 |       387.99 |
+------------+---------+--------------+
21 rows in set (0.09 sec)
```

<1>、<2>、<3>、<4>、<5>处应填的正确值分别为：

1．a

2．a.product_cd

3．product_type_cd

4．p.product_cd

5．a.cust_id

第 4 章

4-1

下面的过滤条件将返回哪些交易的 ID？

```
txn_date < '2005-02-26' AND (txn_type_cd = 'DBT' OR amount > 100)
```

返回的交易 ID 为 1、2、3、5、6 和 7。

4-2

下面的过滤条件将返回哪些交易的 ID?

```
account_id IN (101,103) AND NOT (txn_type_cd = 'DBT' OR amount > 100)
```

返回的交易 ID 为 4 和 9。

4-3

构造查询语句，获取在 2002 年开户的所有账户。

```
mysql> SELECT account_id, open_date
    -> FROM account
    -> WHERE open_date BETWEEN '2002-01-01' AND '2002-12-31';
+------------+------------+
| account_id | open_date  |
+------------+------------+
|          6 | 2002-11-23 |
|          7 | 2002-12-15 |
|         12 | 2002-08-24 |
|         20 | 2002-09-30 |
|         21 | 2002-10-01 |
+------------+------------+
5 rows in set (0.01 sec)
```

4-4

构造查询,查找姓氏中以 a 为第二个字符,并且 e 在 a 后面任意位置出现的非公司顾客。

```
mysql> SELECT cust_id, lname, fname
    -> FROM individual
    -> WHERE lname LIKE '_a%e%';
+---------+--------+---------+
| cust_id | lname  | fname   |
+---------+--------+---------+
|       1 | Hadley | James   |
|       9 | Farley | Richard |
+---------+--------+---------+
2 rows in set (0.02 sec)
```

第 5 章

5-1

给下面的查询填空（使用<#>标记），以获得其后的结果。

```
mysql> SELECT e.emp_id, e.fname, e.lname, b.name
    -> FROM employee e INNER JOIN <1> b
    ->   ON e.assigned_branch_id = b.<2>;
```

```
+--------+----------+-----------+---------------+
| emp_id | fname    | lname     | name          |
+--------+----------+-----------+---------------+
|      1 | Michael  | Smith     | Headquarters  |
|      2 | Susan    | Barker    | Headquarters  |
|      3 | Robert   | Tyler     | Headquarters  |
|      4 | Susan    | Hawthorne | Headquarters  |
|      5 | John     | Gooding   | Headquarters  |
|      6 | Helen    | Fleming   | Headquarters  |
|      7 | Chris    | Tucker    | Headquarters  |
|      8 | Sarah    | Parker    | Headquarters  |
|      9 | Jane     | Grossman  | Headquarters  |
|     10 | Paula    | Roberts   | Woburn Branch |
|     11 | Thomas   | Ziegler   | Woburn Branch |
|     12 | Samantha | Jameson   | Woburn Branch |
|     13 | John     | Blake     | Quincy Branch |
|     14 | Cindy    | Mason     | Quincy Branch |
|     15 | Frank    | Portman   | Quincy Branch |
|     16 | Theresa  | Markham   | So. NH Branch |
|     17 | Beth     | Fowler    | So. NH Branch |
|     18 | Rick     | Tulman    | So. NH Branch |
+--------+----------+-----------+---------------+
18 rows in set (0.03 sec)
```

<1>、<2>处应填的正确值分别为：

1．branch

2．branch_id

5-2

编写查询，返回所有非商务顾客的账户 ID（customer.cust_type_cd = 'I'）、顾客的联邦个人识别号码（cusomer.fed_id）以及账户所依赖的产品名称（product.name）。

```
mysql> SELECT a.account_id, c.fed_id, p.name
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    ->   INNER JOIN product p
    ->   ON a.product_cd = p.product_cd
    -> WHERE c.cust_type_cd = 'I';
+------------+-------------+------------------------+
| account_id | fed_id      | name                   |
+------------+-------------+------------------------+
|          1 | 111-11-1111 | checking account       |
|          2 | 111-11-1111 | savings account        |
|          3 | 111-11-1111 | certificate of deposit |
|          4 | 222-22-2222 | checking account       |
|          5 | 222-22-2222 | savings account        |
|          6 | 333-33-3333 | checking account       |
```

```
|          7 | 333-33-3333 | money market account   |
|          8 | 444-44-4444 | checking account       |
|          9 | 444-44-4444 | savings account        |
|         10 | 444-44-4444 | money market account   |
|         11 | 555-55-5555 | checking account       |
|         12 | 666-66-6666 | checking account       |
|         13 | 666-66-6666 | certificate of deposit |
|         14 | 777-77-7777 | certificate of deposit |
|         15 | 888-88-8888 | checking account       |
|         16 | 888-88-8888 | savings account        |
|         17 | 999-99-9999 | checking account       |
|         18 | 999-99-9999 | money market account   |
|         19 | 999-99-9999 | certificate of deposit |
+------------+-------------+------------------------+
19 rows in set (0.00 sec)
```

5-3

构建查询，查找所有主管位于另一个部门的雇员，需要获取该雇员的 ID、姓氏和名字。

```
mysql> SELECT e.emp_id, e.fname, e.lname
    -> FROM employee e INNER JOIN employee mgr
    ->   ON e.superior_emp_id = mgr.emp_id
    -> WHERE e.dept_id != mgr.dept_id;
+--------+-------+-----------+
| emp_id | fname | lname     |
+--------+-------+-----------+
|      4 | Susan | Hawthorne |
|      5 | John  | Gooding   |
+--------+-------+-----------+
2 rows in set (0.00 sec)
```

第 6 章

6-1

如果集合 A = {L M N O P} ，集合 B = {P Q R S T}，那么下面的操作产生的结果集是什么？

- A union B
- A union all B
- A intersect B
- A except B

1．A union B = {L M N O P Q R S T}

2．A union all B = {L M N O P P Q R S T}

3．A intersect B = {P}

4．A except B = {L M N O}

6-2

编写一个复合查询，以查找所有个人客户以及雇员的姓氏和名字。

```
mysql> SELECT fname, lname
    -> FROM individual
    -> UNION
    -> SELECT fname, lname
    -> FROM employee;
+----------+-----------+
| fname    | lname     |
+----------+-----------+
| James    | Hadley    |
| Susan    | Tingley   |
| Frank    | Tucker    |
| John     | Hayward   |
| Charles  | Frasier   |
| John     | Spencer   |
| Margaret | Young     |
| Louis    | Blake     |
| Richard  | Farley    |
| Michael  | Smith     |
| Susan    | Barker    |
| Robert   | Tyler     |
| Susan    | Hawthorne |
| John     | Gooding   |
| Helen    | Fleming   |
| Chris    | Tucker    |
| Sarah    | Parker    |
| Jane     | Grossman  |
| Paula    | Roberts   |
| Thomas   | Ziegler   |
| Samantha | Jameson   |
| John     | Blake     |
| Cindy    | Mason     |
| Frank    | Portman   |
| Theresa  | Markham   |
| Beth     | Fowler    |
| Rick     | Tulman    |
+----------+-----------+
27 rows in set (0.01 sec)
```

6-3

根据 lname 列对练习 6-2 的结果进行排序。

```
mysql> SELECT fname, lname
    -> FROM individual
    -> UNION ALL
    -> SELECT fname, lname
    -> FROM employee
    -> ORDER BY lname;
+----------+-----------+
| fname    | lname     |
+----------+-----------+
| Susan    | Barker    |
| Louis    | Blake     |
| John     | Blake     |
| Richard  | Farley    |
| Helen    | Fleming   |
| Beth     | Fowler    |
| Charles  | Frasier   |
| John     | Gooding   |
| Jane     | Grossman  |
| James    | Hadley    |
| Susan    | Hawthorne |
| John     | Hayward   |
| Samantha | Jameson   |
| Theresa  | Markham   |
| Cindy    | Mason     |
| Sarah    | Parker    |
| Frank    | Portman   |
| Paula    | Roberts   |
| Michael  | Smith     |
| John     | Spencer   |
| Susan    | Tingley   |
| Chris    | Tucker    |
| Frank    | Tucker    |
| Rick     | Tulman    |
| Robert   | Tyler     |
| Margaret | Young     |
| Thomas   | Ziegler   |
+----------+-----------+
27 rows in set (0.01 sec)
```

第 7 章

7-1

编写查询，返回字符串‘Please find the substring in this string’的第 17 个和第 25 个字符。

```
mysql> SELECT SUBSTRING('Please find the substring in this string',17,9);
+------------------------------------------------------------+
| SUBSTRING('Please find the substring in this string',17,9) |
```

```
+--------------------------------------------------------------+
| substring                                                    |
+--------------------------------------------------------------+
1 row in set (0.00 sec)
```

7-2

编写查询，返回数字-25.76823 的绝对值与符号（-1、0 或 1），并将返回值四舍五入至百分位。

```
mysql> SELECT ABS(-25.76823), SIGN(-25.76823), ROUND(-25.76823, 2);
+----------------+-----------------+---------------------+
| ABS(-25.76823) | SIGN(-25.76823) | ROUND(-25.76823, 2) |
+----------------+-----------------+---------------------+
|       25.76823 |              -1 |              -25.77 |
+----------------+-----------------+---------------------+
1 row in set (0.00 sec)
```

7-3

编写查询，返回当前日期所在的月份。

```
mysql> SELECT EXTRACT(MONTH FROM CURRENT_DATE());
+----------------------------------+
| EXTRACT(MONTH FROM CURRENT_DATE) |
+----------------------------------+
|                                5 |
+----------------------------------+
1 row in set (0.02 sec)
```

（你的查询结果很可能与上面不同，除非正好你也是在 5 月做这个练习题的。）

第 8 章

8-1

构建查询，对 account 表的数据行计数。

```
mysql> SELECT COUNT(*)
    -> FROM account;
+----------+
| count(*) |
+----------+
|       24 |
+----------+
1 row in set (0.32 sec)
```

8-2

修改练习 8-1 中的查询，使之对每个客户所持有的账户计数，并且显示每个客户的 ID

及其账户数。

```
mysql> SELECT cust_id, COUNT(*)
    -> FROM account
    -> GROUP BY cust_id;
+---------+----------+
| cust_id | count(*) |
+---------+----------+
|       1 |        3 |
|       2 |        2 |
|       3 |        2 |
|       4 |        3 |
|       5 |        1 |
|       6 |        2 |
|       7 |        1 |
|       8 |        2 |
|       9 |        3 |
|      10 |        2 |
|      11 |        1 |
|      12 |        1 |
|      13 |        1 |
+---------+----------+
13 rows in set (0.00 sec)
```

8-3

修改练习 8-2 的查询，使之只包含至少持有两个账户的客户。

```
mysql> SELECT cust_id, COUNT(*)
    -> FROM account
    -> GROUP BY cust_id
    -> HAVING COUNT(*) >= 2;
+---------+----------+
| cust_id | COUNT(*) |
+---------+----------+
|       1 |        3 |
|       2 |        2 |
|       3 |        2 |
|       4 |        3 |
|       6 |        2 |
|       8 |        2 |
|       9 |        3 |
|      10 |        2 |
+---------+----------+
8 rows in set (0.04 sec)
```

8-4（附加题）

查找至少包含一个账户的产品和支行组合的可用余额合计数，并根据余额合计数对结果进行排序（从最高到最低）。

```
mysql> SELECT product_cd, open_branch_id, SUM(avail_balance)
    -> FROM account
    -> GROUP BY product_cd, open_branch_id
    -> HAVING COUNT(*) > 1
    -> ORDER BY 3 DESC;
+------------+----------------+--------------------+
| product_cd | open_branch_id | SUM(avail_balance) |
+------------+----------------+--------------------+
| CHK        |              4 |           67852.33 |
| MM         |              1 |           14832.64 |
| CD         |              1 |           11500.00 |
| CD         |              2 |            8000.00 |
| CHK        |              2 |            3315.77 |
| CHK        |              1 |             782.16 |
| SAV        |              2 |             700.00 |
+------------+----------------+--------------------+
7 rows in set (0.01 sec)
```

注意 MySQL 并不接受“ORDER BY SUM(avail_balance) DESC”，因此这里不得不指明排序列的位置。

第 9 章

9-1

对 account 表编写一个查询，过滤条件使用的非关联子查询实现对 product 表查找所有贷款账户（product.product_type_cd = 'LOAN'），结果包括账号 ID、产品代码、客户 ID 和可用余额。

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd IN (SELECT product_cd
    ->   FROM product
    ->   WHERE product_type_cd = 'LOAN');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         21 | BUS        |      10 |          0.00 |
|         22 | BUS        |      11 |       9345.55 |
|         24 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
3 rows in set (0.07 sec)
```

9-2

重做练习 9-1，对 product 表使用关联子查询获得同样的结果。

```
mysql> SELECT a.account_id, a.product_cd, a.cust_id, a.avail_balance
```

```
    -> FROM account a
    -> WHERE EXISTS (SELECT 1
    ->   FROM product p
    ->   WHERE p.product_cd = a.product_cd
    ->     AND p.product_type_cd = 'LOAN');
+------------+------------+---------+--------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+--------------+
|         21 | BUS        |      10 |          0.00 |
|         22 | BUS        |      11 |       9345.55 |
|         24 | SBL        |      13 |      50000.00 |
+------------+------------+---------+--------------+
3 rows in set (0.01 sec)
```

9-3

将下面的查询与 employee 表连接，以展示每个雇员的经验。

```
SELECT 'trainee' name, '2004-01-01' start_dt, '2005-12-31' end_dt
UNION ALL
SELECT 'worker' name, '2002-01-01' start_dt, '2003-12-31' end_dt
UNION ALL
SELECT 'mentor' name, '2000-01-01' start_dt, '2001-12-31' end_dt
```

子查询别名定义为 levels，它包含雇员 ID、名字、姓氏以及经验等级（levels.name）。（提示：利用不等条件构建连接条件，确定 employee.start_date 列位于哪个等级。）

```
mysql> SELECT e.emp_id, e.fname, e.lname, levels.name
    -> FROM employee e INNER JOIN
    ->  (SELECT 'trainee' name, '2004-01-01' start_dt, '2005-12-31' end_dt
    ->   UNION ALL
    ->   SELECT 'worker' name, '2002-01-01' start_dt, '2003-12-31' end_dt
    ->   UNION ALL
    ->   SELECT 'mentor' name, '2000-01-01' start_dt, '2001-12-31' end_dt)
levels
    ->   ON e.start_date BETWEEN levels.start_dt AND levels.end_dt;

+--------+----------+-----------+---------+
| emp_id | fname    | lname     | name    |
+--------+----------+-----------+---------+
|      6 | Helen    | Fleming   | trainee |
|      7 | Chris    | Tucker    | trainee |
|      2 | Susan    | Barker    | worker  |
|      4 | Susan    | Hawthorne | worker  |
|      5 | John     | Gooding   | worker  |
|      8 | Sarah    | Parker    | worker  |
|      9 | Jane     | Grossman  | worker  |
|     10 | Paula    | Roberts   | worker  |
|     12 | Samantha | Jameson   | worker  |
|     14 | Cindy    | Mason     | worker  |
|     15 | Frank    | Portman   | worker  |
|     17 | Beth     | Fowler    | worker  |
```

```
|     18 | Rick     | Tulman    | worker  |
|      1 | Michael  | Smith     | mentor  |
|      3 | Robert   | Tyler     | mentor  |
|     11 | Thomas   | Ziegler   | mentor  |
|     13 | John     | Blake     | mentor  |
|     16 | Theresa  | Markham   | mentor  |
+--------+----------+-----------+---------+
18 rows in set (0.00 sec)
```

9-4

对 employee 构建一个查询，检索雇员 ID、名字、姓氏及其所属部门和分行的名字。请不要连接任何表。

```
mysql> SELECT e.emp_id, e.fname, e.lname,
    ->  (SELECT d.name FROM department d
    ->   WHERE d.dept_id = e.dept_id) dept_name,
    ->  (SELECT b.name FROM branch b
    ->   WHERE b.branch_id = e.assigned_branch_id) branch_name
    -> FROM employee e;
+--------+----------+-----------+----------------+---------------+
| emp_id | fname    | lname     | dept_name      | branch_name   |
+--------+----------+-----------+----------------+---------------+
|      1 | Michael  | Smith     | Administration | Headquarters  |
|      2 | Susan    | Barker    | Administration | Headquarters  |
|      3 | Robert   | Tyler     | Administration | Headquarters  |
|      4 | Susan    | Hawthorne | Operations     | Headquarters  |
|      5 | John     | Gooding   | Loans          | Headquarters  |
|      6 | Helen    | Fleming   | Operations     | Headquarters  |
|      7 | Chris    | Tucker    | Operations     | Headquarters  |
|      8 | Sarah    | Parker    | Operations     | Headquarters  |
|      9 | Jane     | Grossman  | Operations     | Headquarters  |
|     10 | Paula    | Roberts   | Operations     | Woburn Branch |
|     11 | Thomas   | Ziegler   | Operations     | Woburn Branch |
|     12 | Samantha | Jameson   | Operations     | Woburn Branch |
|     13 | John     | Blake     | Operations     | Quincy Branch |
|     14 | Cindy    | Mason     | Operations     | Quincy Branch |
|     15 | Frank    | Portman   | Operations     | Quincy Branch |
|     16 | Theresa  | Markham   | Operations     | So. NH Branch |
|     17 | Beth     | Fowler    | Operations     | So. NH Branch |
|     18 | Rick     | Tulman    | Operations     | So. NH Branch |
+--------+----------+-----------+----------------+---------------+
18 rows in set (0.12 sec)
```

第 10 章

10-1

编写一个查询，它返回所有产品名称及基于该产品的账号(用 account 表里的 product_cd

列连接 product 表)。查询结果需要包括所有产品，即使这个产品并没有客户开户。

```
mysql> SELECT p.product_cd, a.account_id, a.cust_id, a.avail_balance
    -> FROM product p LEFT OUTER JOIN account a
    ->   ON p.product_cd = a.product_cd;
+------------+------------+---------+---------------+
| product_cd | account_id | cust_id | avail_balance |
+------------+------------+---------+---------------+
| AUT        |       NULL |    NULL |          NULL |
| BUS        |         21 |      10 |          0.00 |
| BUS        |         22 |      11 |       9345.55 |
| CD         |          3 |       1 |       3000.00 |
| CD         |         13 |       6 |      10000.00 |
| CD         |         14 |       7 |       5000.00 |
| CD         |         19 |       9 |       1500.00 |
| CHK        |          1 |       1 |       1057.75 |
| CHK        |          4 |       2 |       2258.02 |
| CHK        |          6 |       3 |       1057.75 |
| CHK        |          8 |       4 |        534.12 |
| CHK        |         11 |       5 |       2237.97 |
| CHK        |         12 |       6 |        122.37 |
| CHK        |         15 |       8 |       3487.19 |
| CHK        |         17 |       9 |        125.67 |
| CHK        |         20 |      10 |      23575.12 |
| CHK        |         23 |      12 |      38552.05 |
| MM         |          7 |       3 |       2212.50 |
| MM         |         10 |       4 |       5487.09 |
| MM         |         18 |       9 |       9345.55 |
| MRT        |       NULL |    NULL |          NULL |
| SAV        |          2 |       1 |        500.00 |
| SAV        |          5 |       2 |        200.00 |
| SAV        |          9 |       4 |        767.77 |
| SAV        |         16 |       8 |        387.99 |
| SBL        |         24 |      13 |      50000.00 |
+------------+------------+---------+---------------+
26 rows in set (0.01 sec)
```

10-2

利用其他外连接类型重写练习 10-1 的查询（比如，若在练习 10-1 中使用了左外连接，这次就使用右外连接），要求查询结果相同。

```
mysql> SELECT p.product_cd, a.account_id, a.cust_id, a.avail_balance
    -> FROM account a RIGHT OUTER JOIN product p
    ->   ON p.product_cd = a.product_cd;
+------------+------------+---------+---------------+
| product_cd | account_id | cust_id | avail_balance |
+------------+------------+---------+---------------+
| AUT        |       NULL |    NULL |          NULL |
| BUS        |         21 |      10 |          0.00 |
```

```
| BUS        |         22 |      11 |       9345.55 |
| CD         |          3 |       1 |       3000.00 |
| CD         |         13 |       6 |      10000.00 |
| CD         |         14 |       7 |       5000.00 |
| CD         |         19 |       9 |       1500.00 |
| CHK        |          1 |       1 |       1057.75 |
| CHK        |          4 |       2 |       2258.02 |
| CHK        |          6 |       3 |       1057.75 |
| CHK        |          8 |       4 |        534.12 |
| CHK        |         11 |       5 |       2237.97 |
| CHK        |         12 |       6 |        122.37 |
| CHK        |         15 |       8 |       3487.19 |
| CHK        |         17 |       9 |        125.67 |
| CHK        |         20 |      10 |      23575.12 |
| CHK        |         23 |      12 |      38552.05 |
| MM         |          7 |       3 |       2212.50 |
| MM         |         10 |       4 |       5487.09 |
| MM         |         18 |       9 |       9345.55 |
| MRT        |       NULL |    NULL |          NULL |
| SAV        |          2 |       1 |        500.00 |
| SAV        |          5 |       2 |        200.00 |
| SAV        |          9 |       4 |        767.77 |
| SAV        |         16 |       8 |        387.99 |
| SBL        |         24 |      13 |      50000.00 |
+------------+------------+---------+---------------+
26 rows in set (0.02 sec)
```

10-3

编写一个查询,将 account 表与 individual 和 business 两个表外连接(通过 account.cust_id 列)。要求结果集中每个账户一行，查询的列有 account.account_id、account.product_cd、individual.fname、individual.lname 和 business.name。

```
mysql> SELECT a.account_id, a.product_cd,
    ->   i.fname, i.lname, b.name
    -> FROM account a LEFT OUTER JOIN business b
    ->   ON a.cust_id = b.cust_id
    ->   LEFT OUTER JOIN individual i
    ->   ON a.cust_id = i.cust_id;
+------------+------------+---------+---------+----------------------+
| account_id | product_cd | fname   | lname   | name                 |
+------------+------------+---------+---------+----------------------+
|          1 | CHK        | James   | Hadley  | NULL                 |
|          2 | SAV        | James   | Hadley  | NULL                 |
|          3 | CD         | James   | Hadley  | NULL                 |
|          4 | CHK        | Susan   | Tingley | NULL                 |
|          5 | SAV        | Susan   | Tingley | NULL                 |
|          6 | CHK        | Frank   | Tucker  | NULL                 |
|          7 | MM         | Frank   | Tucker  | NULL                 |
```

```
|          8 | CHK        | John     | Hayward | NULL                 |
|          9 | SAV        | John     | Hayward | NULL                 |
|         10 | MM         | John     | Hayward | NULL                 |
|         11 | CHK        | Charles  | Frasier | NULL                 |
|         12 | CHK        | John     | Spencer | NULL                 |
|         13 | CD         | John     | Spencer | NULL                 |
|         14 | CD         | Margaret | Young   | NULL                 |
|         15 | CHK        | Louis    | Blake   | NULL                 |
|         16 | SAV        | Louis    | Blake   | NULL                 |
|         17 | CHK        | Richard  | Farley  | NULL                 |
|         18 | MM         | Richard  | Farley  | NULL                 |
|         19 | CD         | Richard  | Farley  | NULL                 |
|        20  | CHK        | NULL     | NULL    |Chilton Engineering   |
|        21  | BUS        | NULL     | NULL    |Chilton Engineering   |
|        22  | BUS        | NULL     | NULL    |Northeast Cooling Inc. |
|        23  | CHK        | NULL     | NULL    |Superior Auto Body    |
|        24  | SBL        | NULL     | NULL    |AAA Insurance Inc.    |
+------------+------------+----------+---------+----------------------+
24 rows in set (0.05 sec)
```

10-4（附加题）

设计一个查询，生成集合{1, 2, 3,⋯, 99, 100}。（提示：应用交叉连接，至少有两个 from 子句的子查询。）

```
SELECT ones.× + tens.× + 1
FROM
 (SELECT 0 × UNION ALL
  SELECT 1 × UNION ALL
  SELECT 2 × UNION ALL
  SELECT 3 × UNION ALL
  SELECT 4 × UNION ALL
  SELECT 5 × UNION ALL
  SELECT 6 × UNION ALL
  SELECT 7 × UNION ALL
  SELECT 8 × UNION ALL
  SELECT 9 ×) ones
CROSS JOIN
(SELECT 0 × UNION ALL
  SELECT 10 × UNION ALL
  SELECT 20 × UNION ALL
  SELECT 30 × UNION ALL
  SELECT 40 × UNION ALL
  SELECT 50 × UNION ALL
  SELECT 60 × UNION ALL
  SELECT 70 × UNION ALL
  SELECT 80 × UNION ALL
  SELECT 90 ×) tens;
```

第 11 章

11-1

重写下面的查询，要求使用查找型 case 表达式替换简单 case 表达式，并且查询结果相同。请读者尽可能少使用 when 子句。

```
SELECT emp_id,
  CASE title
    WHEN 'President' THEN 'Management'
    WHEN 'Vice President' THEN 'Management'
    WHEN 'Treasurer' THEN 'Management'
    WHEN 'Loan Manager' THEN 'Management'
    WHEN 'Operations Manager' THEN 'Operations'
    WHEN 'Head Teller' THEN 'Operations'
    WHEN 'Teller' THEN 'Operations'
    ELSE 'Unknown'
  END
FROM employee;

SELECT emp_id,
  CASE
    WHEN title LIKE '%President' OR title = 'Loan Manager'
      OR title = 'Treasurer'
      THEN 'Management'
    WHEN title LIKE '%Teller' OR title = 'Operations Manager'
      THEN 'Operations'
    ELSE 'Unknown'
  END
FROM employee;
```

11-2

重写下面的查询,要求结果集为单行 4 列(每个分行 1 列)的,其中 4 列分别以 branch_1～branch_4 命名。

```
mysql> SELECT open_branch_id, COUNT(*)
    -> FROM account
    -> GROUP BY open_branch_id;
+----------------+----------+
| open_branch_id | COUNT(*) |
+----------------+----------+
|              1 |        8 |
|              2 |        7 |
|              3 |        3 |
|              4 |        6 |
+----------------+----------+
4 rows in set (0.00 sec)
mysql> SELECT
    ->   SUM(CASE WHEN open_branch_id = 1 THEN 1 ELSE 0 END) branch_1,
```

```
    ->   SUM(CASE WHEN open_branch_id = 2 THEN 1 ELSE 0 END) branch_2,
    ->   SUM(CASE WHEN open_branch_id = 3 THEN 1 ELSE 0 END) branch_3,
    ->   SUM(CASE WHEN open_branch_id = 4 THEN 1 ELSE 0 END) branch_4
    -> FROM account;
+----------+----------+----------+----------+
| branch_1 | branch_2 | branch_3 | branch_4 |
+----------+----------+----------+----------+
|        8 |        7 |        3 |        6 |
+----------+----------+----------+----------+
1 row in set (0.02 sec)
```

第 12 章

12-1

生成一个事务，它从 Frank Tucker 的货币市场账户存款转账$50 到他的支票账户。要求插入两行到 transaction 并更新 account 表中相应的两行内容。

```
START TRANSACTION;

SELECT i.cust_id,
 (SELECT a.account_id FROM account a
  WHERE a.cust_id = i.cust_id
    AND a.product_cd = 'MM') mm_id,
 (SELECT a.account_id FROM account a
  WHERE a.cust_id = i.cust_id
    AND a.product_cd = 'chk') chk_id
INTO @cst_id, @mm_id, @chk_id
FROM individual i
WHERE i.fname = 'Frank' AND i.lname = 'Tucker';

INSERT INTO transaction (txn_id, txn_date, account_id,
  txn_type_cd, amount)
VALUES (NULL, now(), @mm_id, 'CDT', 50);

INSERT INTO transaction (txn_id, txn_date, account_id,
  txn_type_cd, amount)
VALUES (NULL, now(), @chk_id, 'DBT', 50);

UPDATE account
SET last_activity_date = now(),
  avail_balance = avail_balance - 50
WHERE account_id = @mm_id;

UPDATE account
SET last_activity_date = now(),
  avail_balance = avail_balance + 50
WHERE account_id = @chk_id;

COMMIT;
```

第 13 章

13-1

修改 account 表，使客户不能在任何产品中拥有多个账户（最多一个）。

```
ALTER TABLE account
ADD CONSTRAINT account_unq1 UNIQUE (cust_id, product_cd);
```

13-2

为 transaction 表生成多列索引，该索引可用于如下两个查询。

```
SELECT txn_date, account_id, txn_type_cd, amount
FROM transaction
WHERE txn_date > cast('2008-12-31 23:59:59' as datetime);

SELECT txn_date, account_id, txn_type_cd, amount
FROM transaction
WHERE txn_date > cast('2008-12-31 23:59:59' as datetime)
  AND amount < 1000;

CREATE INDEX txn_idx01
ON transaction (txn_date, amount);
```

第 14 章

14-1

创建一个视图，查询 employee 表并生成下列结果，要求不使用 where 子句。

```
+-----------------+------------------+
| supervisor_name | employee_name    |
+-----------------+------------------+
| NULL            | Michael Smith    |
| Michael Smith   | Susan Barker     |
| Michael Smith   | Robert Tyler     |
| Robert Tyler    | Susan Hawthorne  |
| Susan Hawthorne | John Gooding     |
| Susan Hawthorne | Helen Fleming    |
| Helen Fleming   | Chris Tucker     |
| Helen Fleming   | Sarah Parker     |
| Helen Fleming   | Jane Grossman    |
| Susan Hawthorne | Paula Roberts    |
| Paula Roberts   | Thomas Ziegler   |
| Paula Roberts   | Samantha Jameson |
| Susan Hawthorne | John Blake       |
| John Blake      | Cindy Mason      |
| John Blake      | Frank Portman    |
| Susan Hawthorne | Theresa Markham  |
```

```
| Theresa Markham | Beth Fowler      |
| Theresa Markham | Rick Tulman      |
+-----------------+------------------+
18 rows in set (1.47 sec)

mysql> CREATE VIEW supervisor_vw
    -> (supervisor_name,
    ->  employee_name
    -> )
    -> AS
    -> SELECT concat(spr.fname, ' ', spr.lname),
    ->  concat(emp.fname, ' ', emp.lname)
    -> FROM employee emp LEFT OUTER JOIN employee spr
    ->  ON emp.superior_emp_id = spr.emp_id;
Query OK, 0 rows affected (0.12 sec)

mysql> SELECT * FROM supervisor_vw;
+-----------------+------------------+
| supervisor_name | employee_name    |
+-----------------+------------------+
| NULL            | Michael Smith    |
| Michael Smith   | Susan Barker     |
| Michael Smith   | Robert Tyler     |
| Robert Tyler    | Susan Hawthorne  |
| Susan Hawthorne | John Gooding     |
| Susan Hawthorne | Helen Fleming    |
| Helen Fleming   | Chris Tucker     |
| Helen Fleming   | Sarah Parker     |
| Helen Fleming   | Jane Grossman    |
| Susan Hawthorne | Paula Roberts    |
| Paula Roberts   | Thomas Ziegler   |
| Paula Roberts   | Samantha Jameson |
| Susan Hawthorne | John Blake       |
| John Blake      | Cindy Mason      |
| John Blake      | Frank Portman    |
| Susan Hawthorne | Theresa Markham  |
| Theresa Markham | Beth Fowler      |
| Theresa Markham | Rick Tulman      |
+-----------------+------------------+
18 rows in set (0.17 sec)
```

14-2

除了查询各分行开立的所有账户的余额，银行总裁还想要一张显示各分行名字及城市的报表。创建一个生成这些数据的视图。

```
mysql> CREATE VIEW branch_summary_vw
    -> (branch_name,
    ->  branch_city,
    ->  total_balance
    -> )
    -> AS
    -> SELECT b.name, b.city, sum(a.avail_balance)
```

```
    -> FROM branch b INNER JOIN account a
    ->   ON b.branch_id = a.open_branch_id
    -> GROUP BY b.name, b.city;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM branch_summary_vw;
+---------------+-------------+---------------+
| branch_name   | branch_city | total_balance |
+---------------+-------------+---------------+
| Headquarters  | Waltham     |      27882.57 |
| Quincy Branch | Quincy      |      53270.25 |
| So. NH Branch | Salem       |      68240.32 |
| Woburn Branch | Woburn      |      21361.32 |
+---------------+-------------+---------------+
4 rows in set (0.01 sec)
```

第 15 章

15-1

编写一个查询，列出 bank 列中的所有索引，要求结果包括表名。

```
mysql> SELECT DISTINCT table_name, index_name
    -> FROM information_schema.statistics
    -> WHERE table_schema = 'bank';
+--------------+--------------------+
| table_name   | index_name         |
+--------------+--------------------+
| account      | PRIMARY            |
| account      | account_unq1       |
| account      | fk_product_cd      |
| account      | fk_a_branch_id     |
| account      | fk_a_emp_id        |
| account      | acc_bal_idx        |
| branch       | PRIMARY            |
| business     | PRIMARY            |
| customer     | PRIMARY            |
| department   | PRIMARY            |
| department   | dept_name_idx      |
| employee     | PRIMARY            |
| employee     | fk_dept_id         |
| employee     | fk_e_branch_id     |
| employee     | fk_e_emp_id        |
| individual   | PRIMARY            |
| officer      | PRIMARY            |
| officer      | fk_o_cust_id       |
| product      | PRIMARY            |
| product      | fk_product_type_cd |
| product_type | PRIMARY            |
```

```
| transaction  | PRIMARY            |
| transaction  | fk_t_account_id    |
| transaction  | fk_teller_emp_id   |
| transaction  | fk_exec_branch_id  |
| transaction  | txn_idx01          |
+--------------+--------------------+
26 rows in set (0.00 sec)
```

15-2

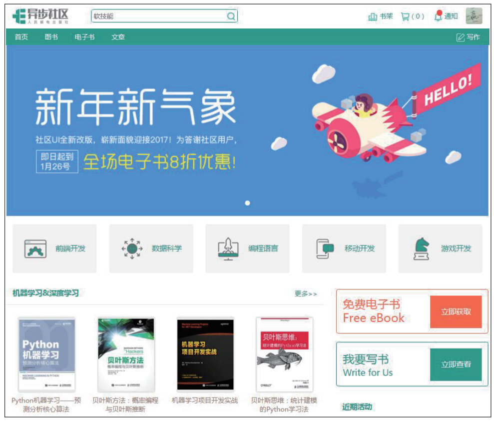
编写一个查询，生成的结果可以用于创建 bank.employee 表的所有索引。要求结果形式如下：

```
"ALTER TABLE <table_name> ADD INDEX <index_name> (<column_list>)"

mysql> SELECT concat(
    ->   CASE
    ->     WHEN st.seq_in_index = 1 THEN
    ->       concat('ALTER TABLE ', st.table_name, ' ADD',
    ->         CASE
    ->           WHEN st.non_unique = 0 THEN ' UNIQUE '
    ->           ELSE ' '
    ->         END,
    ->         'INDEX ',
    ->         st.index_name, ' (', st.column_name)
    ->     ELSE concat('  ', st.column_name)
    ->   END,
    ->   CASE
    ->     WHEN st.seq_in_index =
    ->      (SELECT max(st2.seq_in_index)
    ->       FROM information_schema.statistics st2
    ->       WHERE st2.table_schema = st.table_schema
    ->         AND st2.table_name = st.table_name
    ->         AND st2.index_name = st.index_name)
    ->       THEN ');'
    ->     ELSE ''
    ->   END
    ->  ) index_creation_statement
    -> FROM information_schema.statistics st
    -> WHERE st.table_schema = 'bank'
    ->   AND st.table_name = 'employee'
    -> ORDER BY st.index_name, st.seq_in_index;
+--------------------------------------------------------------------+
| index_creation_statement                                           |
+--------------------------------------------------------------------+
| ALTER TABLE employee ADD INDEX fk_dept_id (dept_id);               |
| ALTER TABLE employee ADD INDEX fk_e_branch_id (assigned_branch_id);|
| ALTER TABLE employee ADD INDEX fk_e_emp_id (superior_emp_id);      |
| ALTER TABLE employee ADD UNIQUE INDEX PRIMARY (emp_id);            |
+--------------------------------------------------------------------+
4 rows in set (0.20 sec)
```

欢迎来到异步社区！

异步社区 人民邮电出版社 www.epubit.com.cn



异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题 可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户账户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 0 使用积分 里填入可使用的积分数值，即可扣减相应金额。

O'REILLY®

SQL学习指南（第2版）（修订版）



无论你需要编写数据库应用程序还是执行数据库管理任务，或是生成数据报表，本书都能够帮助你轻松掌握SQL语言的基础知识。

本书教你学会以下技能：

- 掌握SQL语言的基础知识和高级特性；
- 使用SQL数据语言创建、操作和获取数据；
- 使用SQL方案语言创建数据库对象，如表、索引和约束；
- 了解数据集如何与查询语句交互，理解子查询的重要性；
- 使用SQL内建函数转换和操作数据，在数据语句中使用条件逻辑。

“如果你决定开始学习SQL语言，那么请卷起袖子大干一场吧，不过别忘了让本书成为你的伙伴。阅读本书并完成书中每个实践练习，可以为创建基于数据库的解决方案做好准备。数据库无所不在，本书向你提供作者在工作中经过实践检验的宝贵经验。”

——Roy Owens
来自CBORD Group公司
的数据库专家

Alan Beaulieu从事设计、构建和实现应用数据库已有15个年头，他目前经营自己的顾问公司，专门提供金融和电信领域的Oracle数据库设计与支持服务。Alan毕业于康奈尔大学工程学院。

www.oreilly.com

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China
(excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/数据库/SQL
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-38344-0
9 787115 383440 >